# Queues

- Reading: Lewis and Loftus, JAVA: Software Solutions, (3rd ed), Chapter 12.2
- Savitch Chapter 10.2

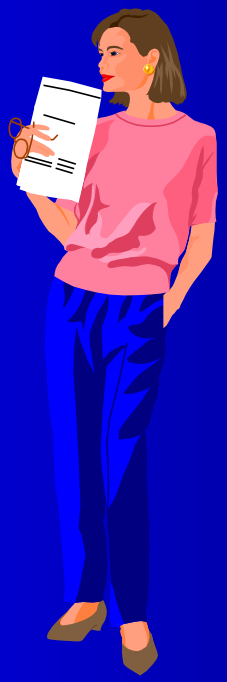
# Objectives

- To introduce the definition of a queue
- To learn how to implement a queue using a linked list
- To develop an implementation of a queue using an array

# Queues

# What is a queue?

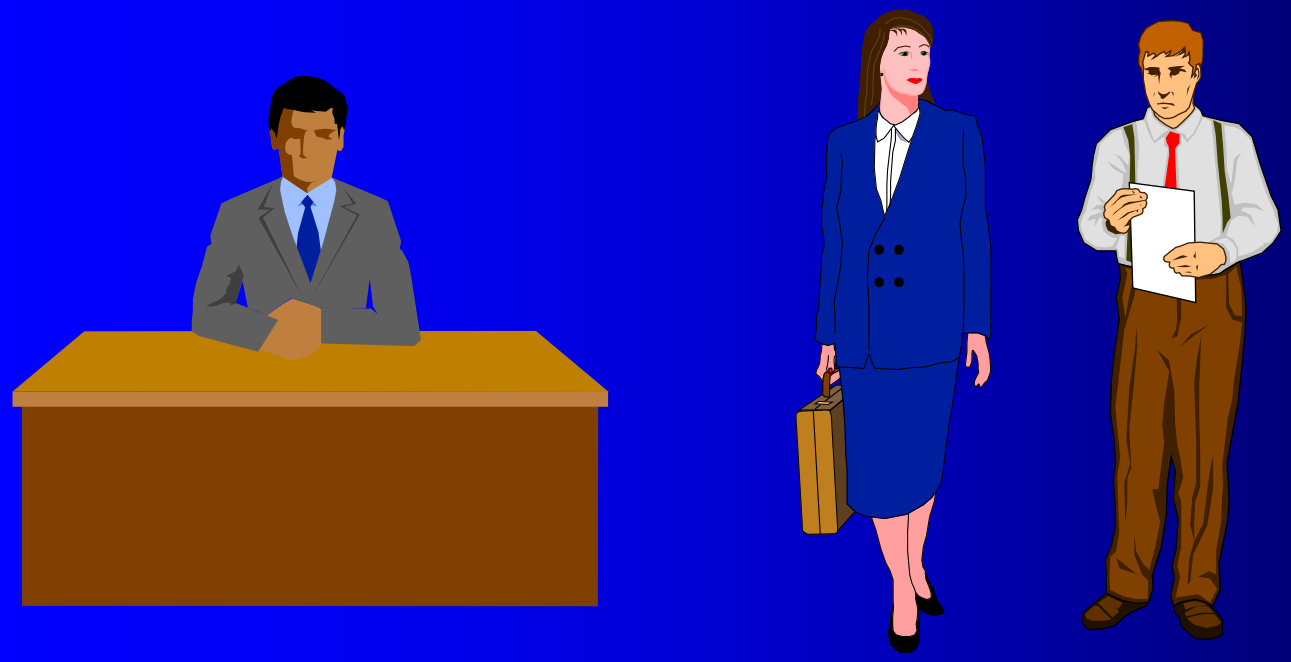
# What is a queue?



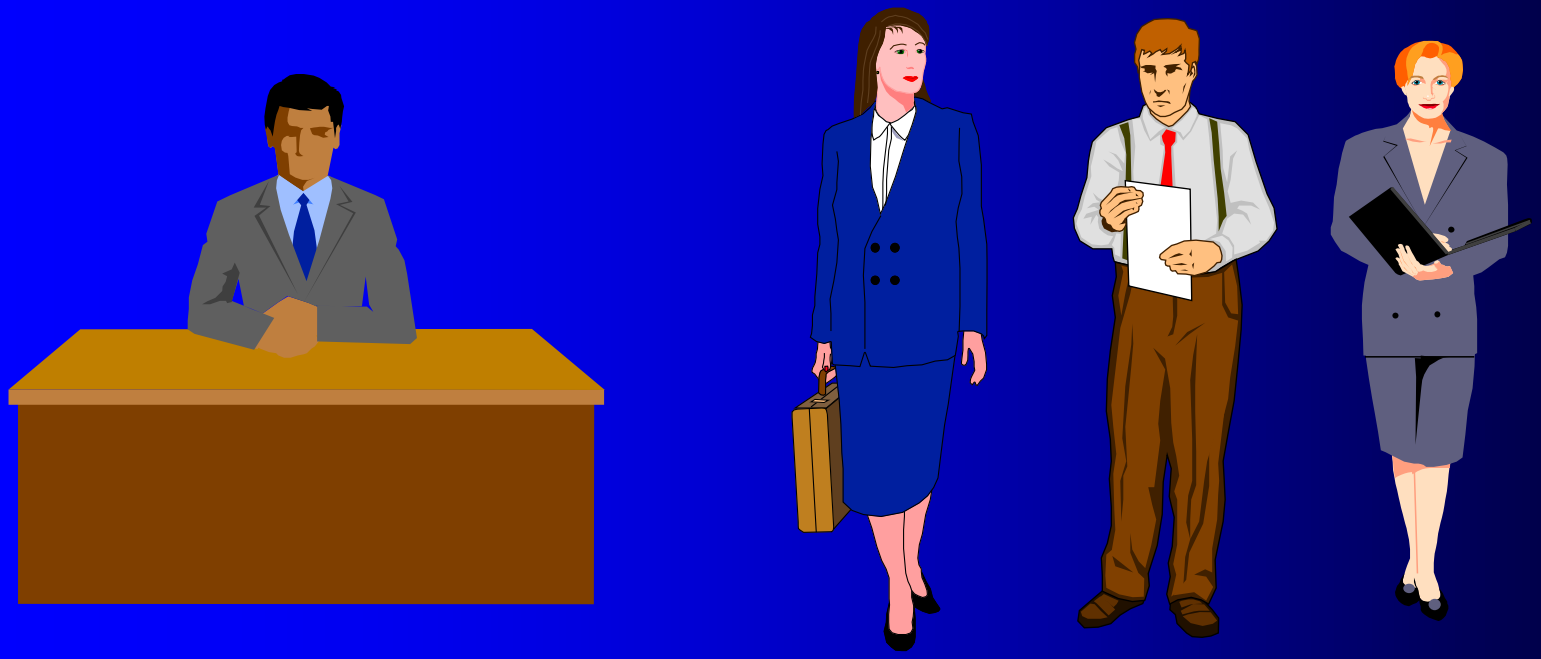
# What is a queue?

# What is a queue?

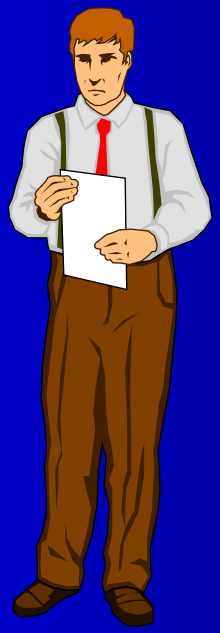
# What is a queue?



# What is a queue?



# What is a queue?



# What is a queue?

# What is a queue?

# Definition of a queue

- A list in which cells may be removed only from the front of the list and new cells may be added only to the end of the list
- Called a FIFO list (“first-in, first-out”)

# Queue Operations

- **insert** to add an element at the rear of the queue
- **get** to get the value and delete the element at the front of the queue
- **isEmpty** to return true if the queue has no element
- **isFull** to return true if the queue is full (array implementation)

# Class exercise: queues

- What is output by the following code?

```
int x;
Queue q;

q.insert (8);
q.insert (9);
q.insert (3);
x = q.get ();
q.insert (18);
x = q.get ();
q.insert (22);
while (! q.isEmpty())
    System.out.println(q.get());
System.out.println(x);
```

# Linked list implementation

- JAVA declaration

```
class QueueNode {
    Object data;
    QueueNode next;
    QueueNode(Object _d) {
            data = _d;
            next = null;
    }
... ...
}
```

Variables are used with package access, just for this teaching session, normally these variables would have private access

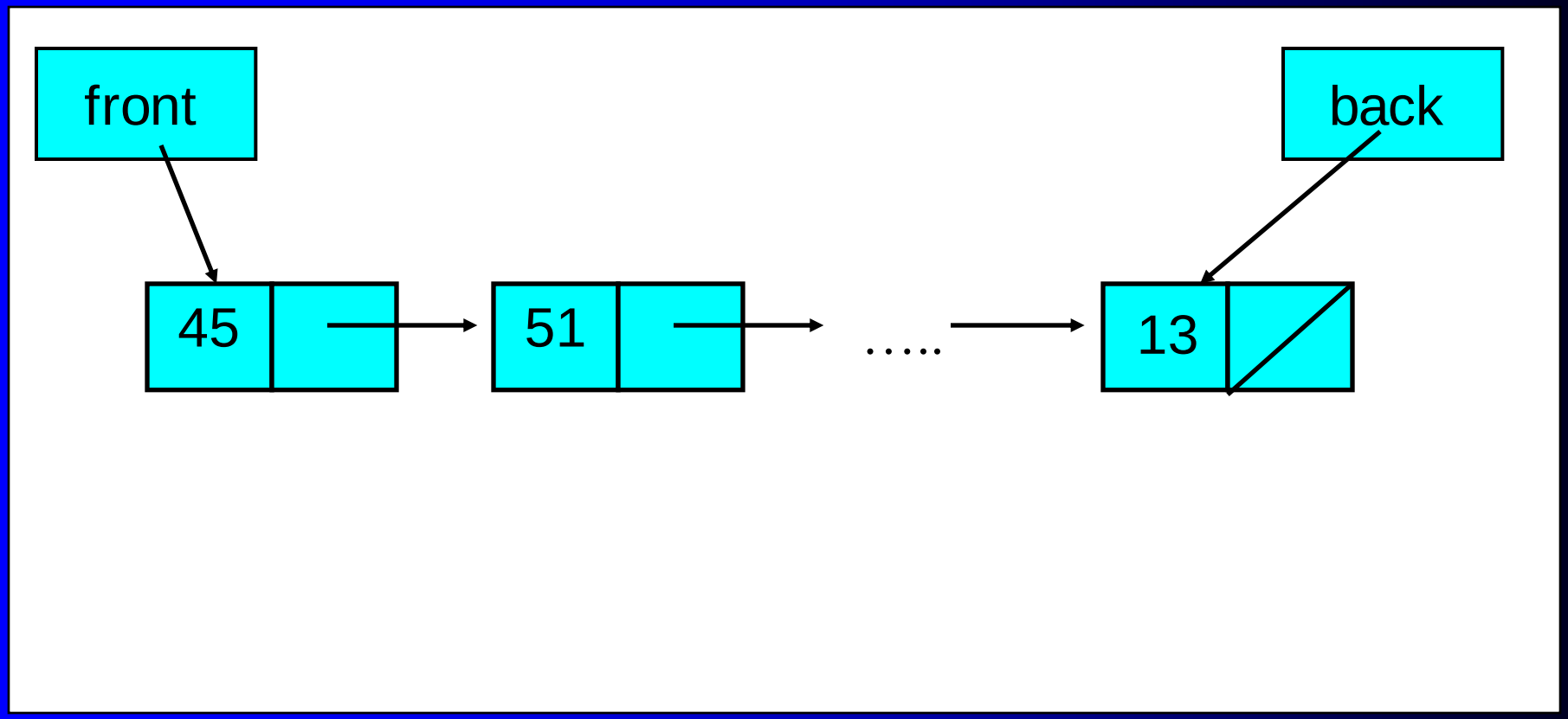
# Linked list implementation

```
public class Queue {
    private QueueNode front = null;
    private QueueNode  back = null;
    public void insert(Object new_data) {... ...}
    public Object get() {... ... }
    public boolean isEmpty() {... ...}
}
```

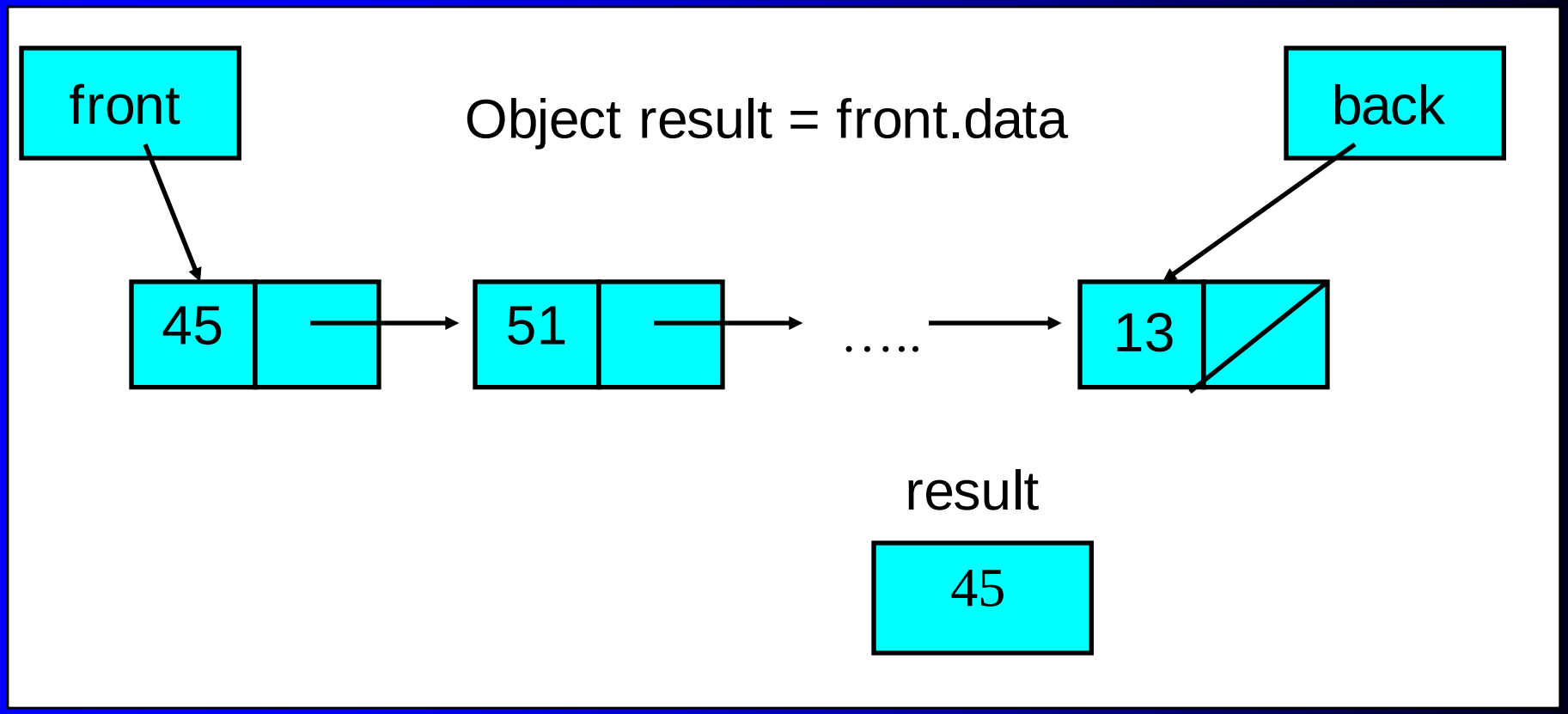
# Linked list implementation

- As with StackNode and Stack, QueueNode may be defined as an inner class within the class Queue.

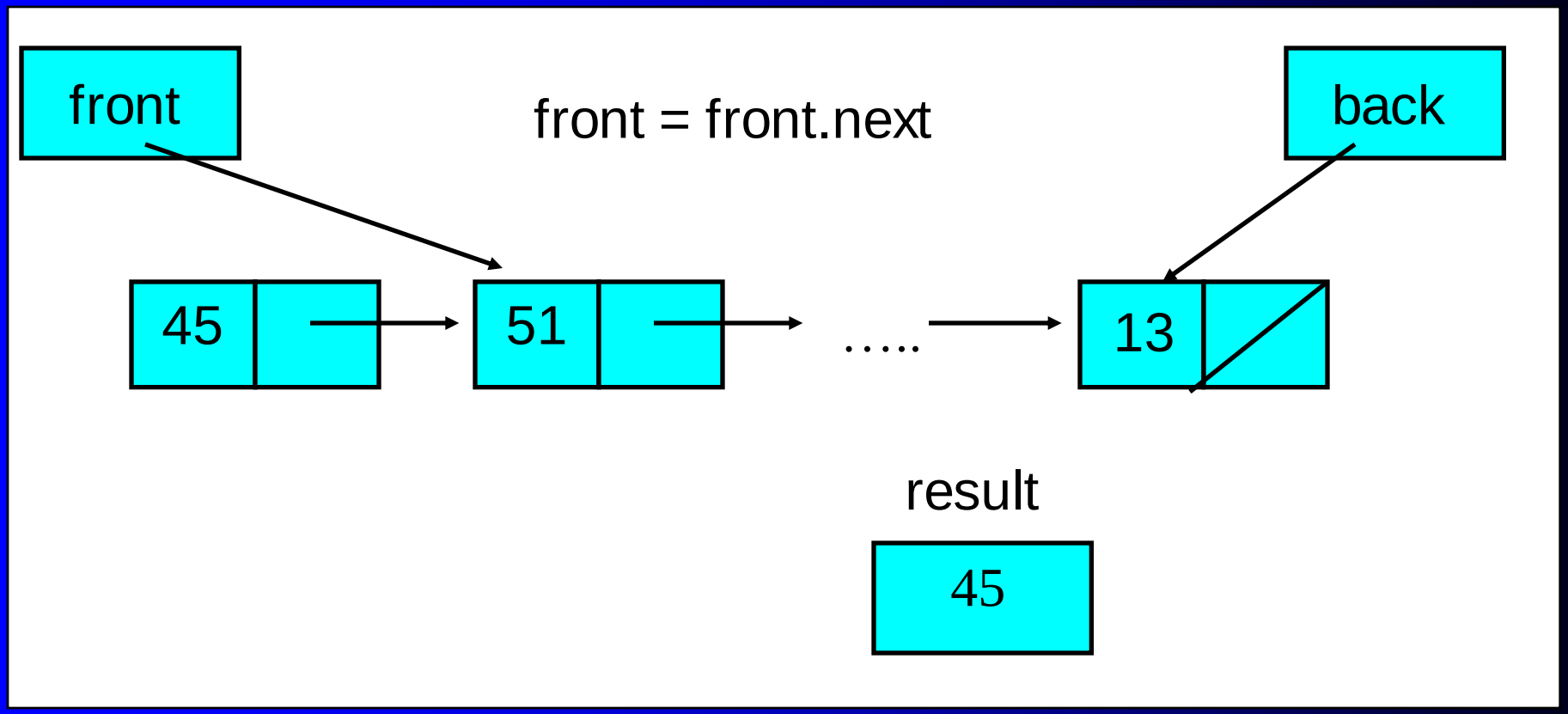
# Linked list implementation



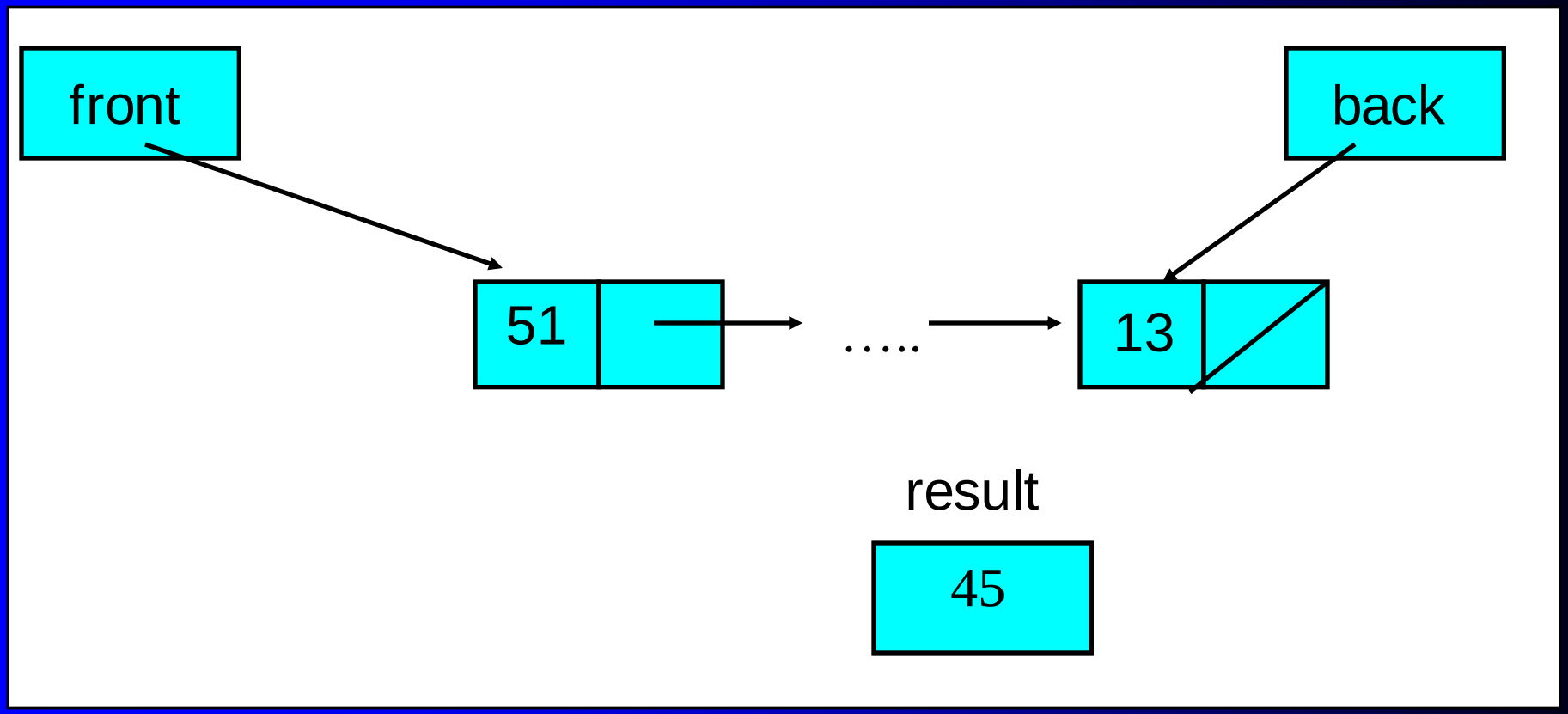
# Linked list implementation - get



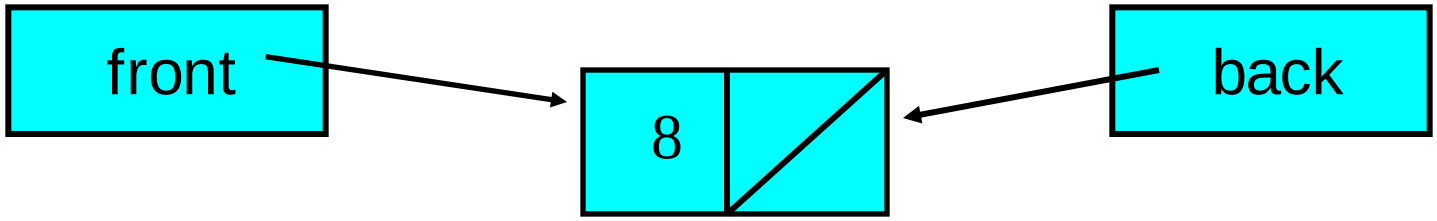
# Linked list implementation - get



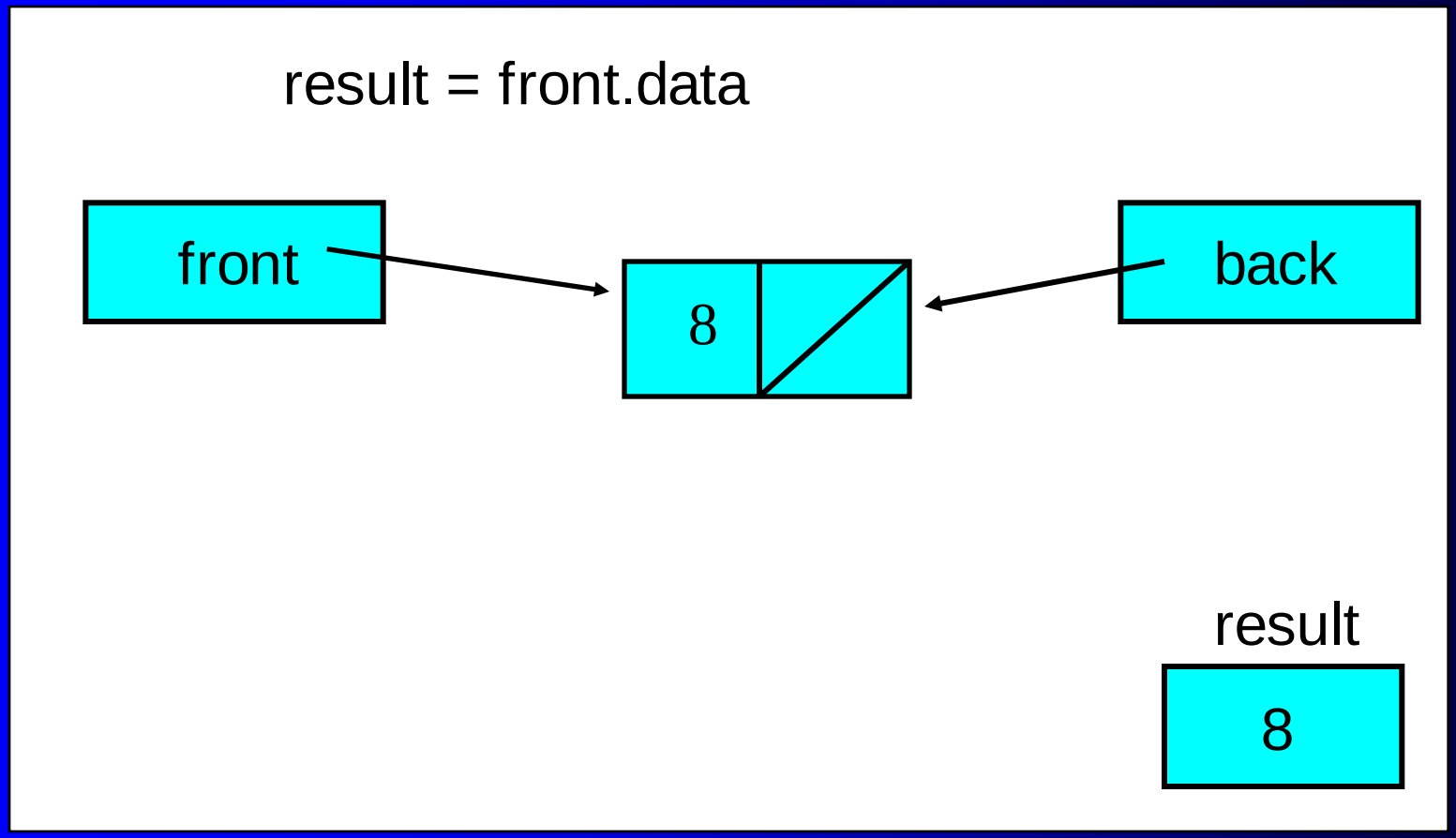
# Linked list implementation - get



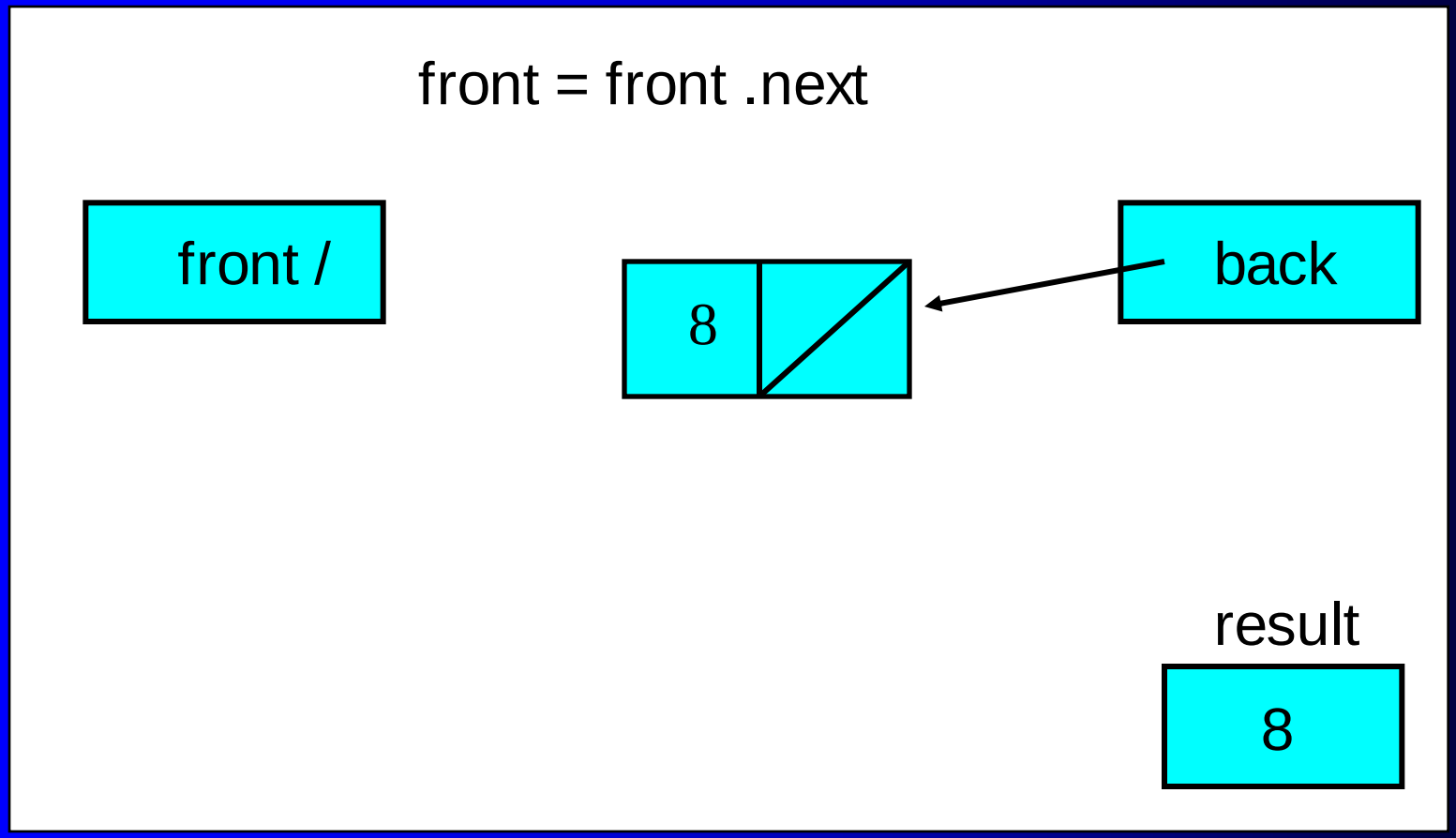
# Get from a singleton queue



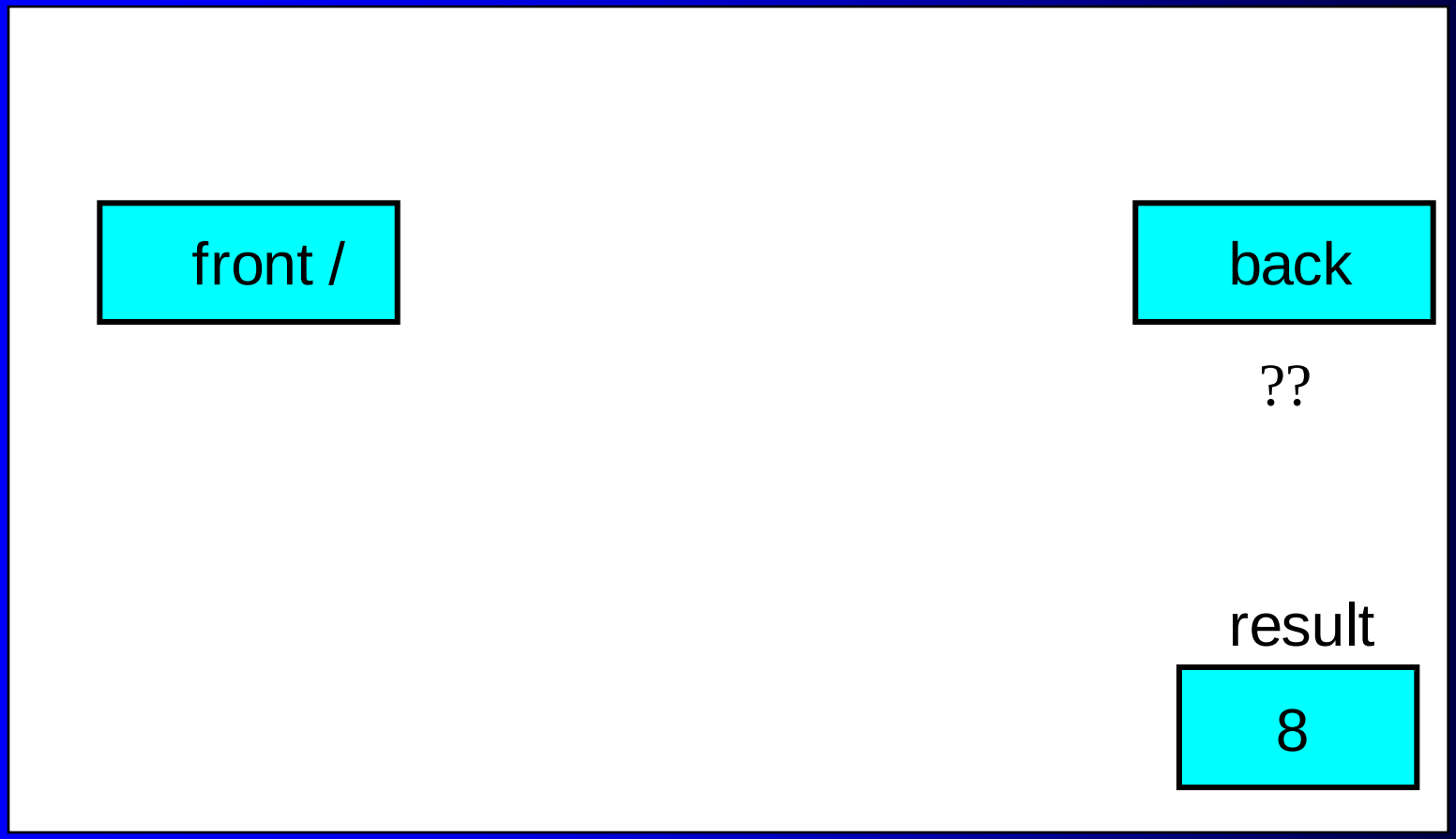
# Get from a singleton queue



# Get from a singleton queue



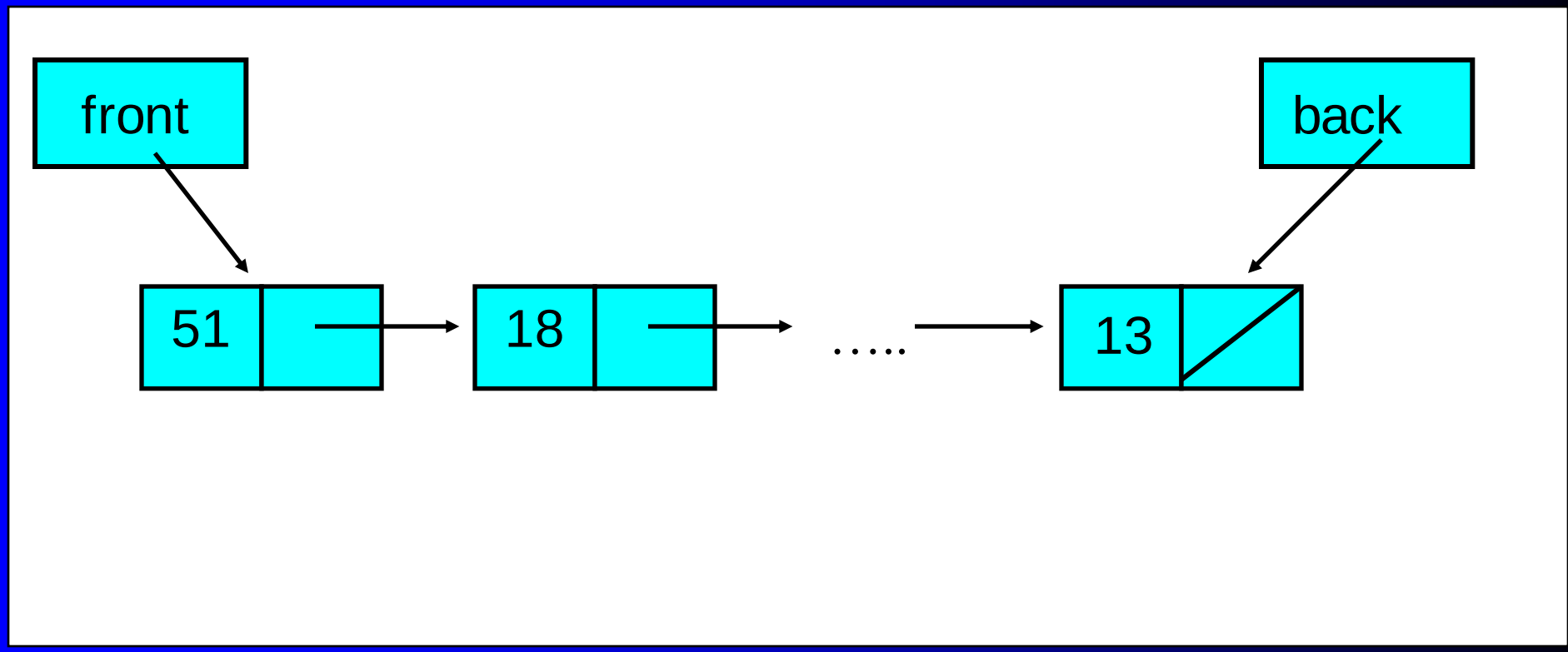
# Get from a singleton queue



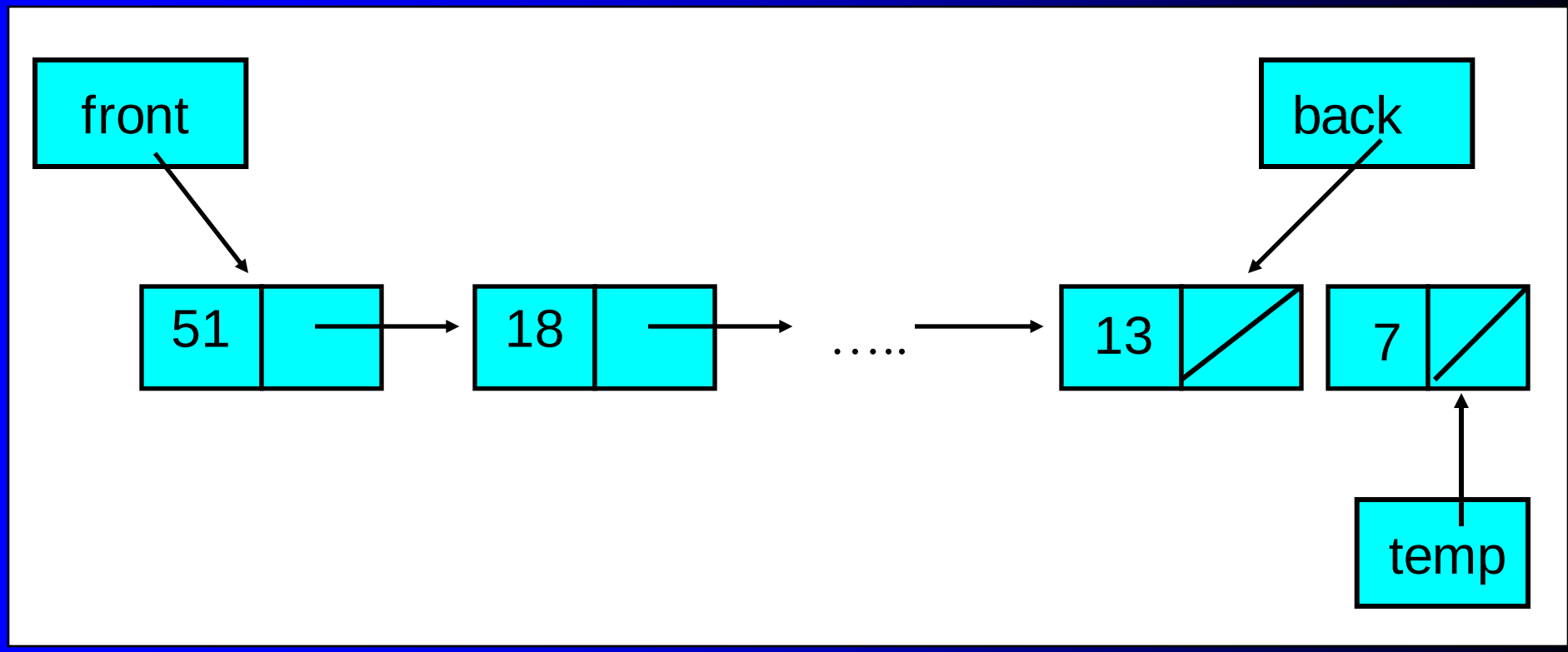
# Special case

```
front = front.next;

if (front == null)
    back = null;
```

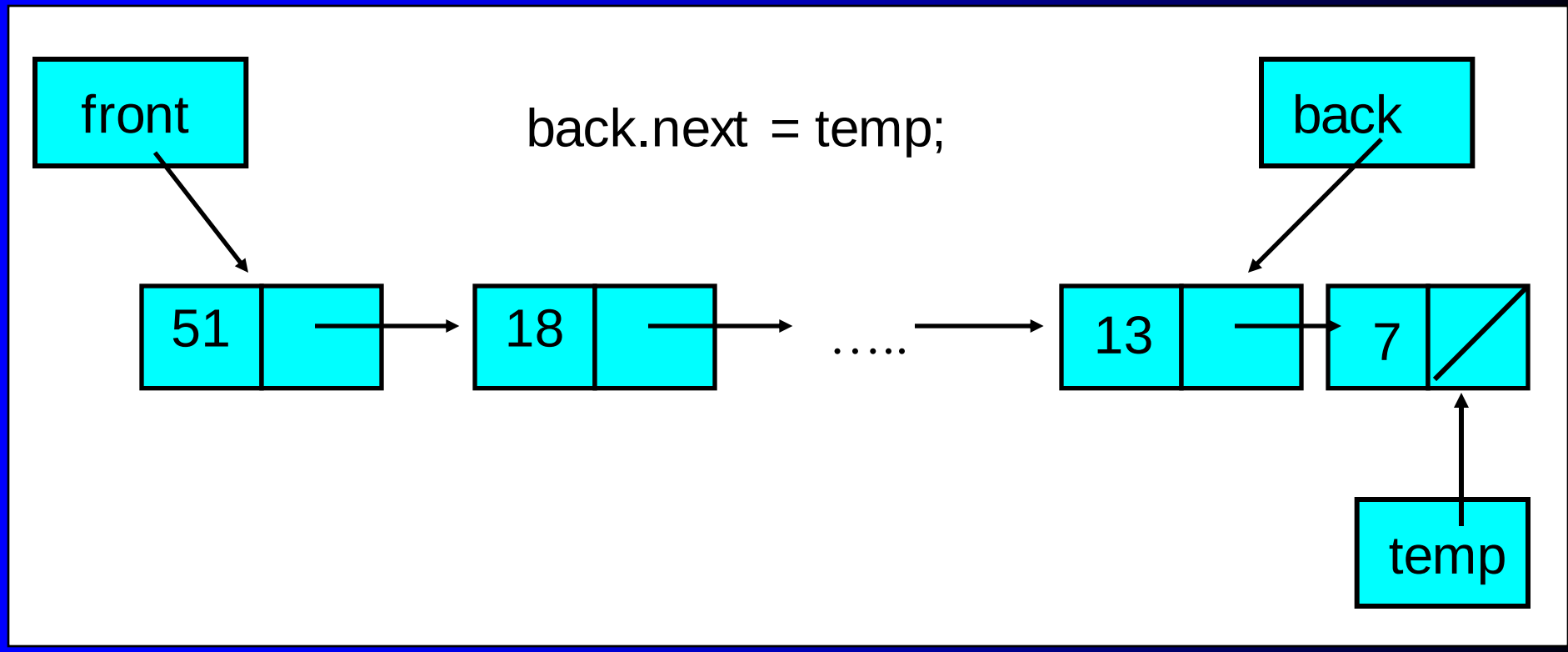
# Linked list implementation - insert



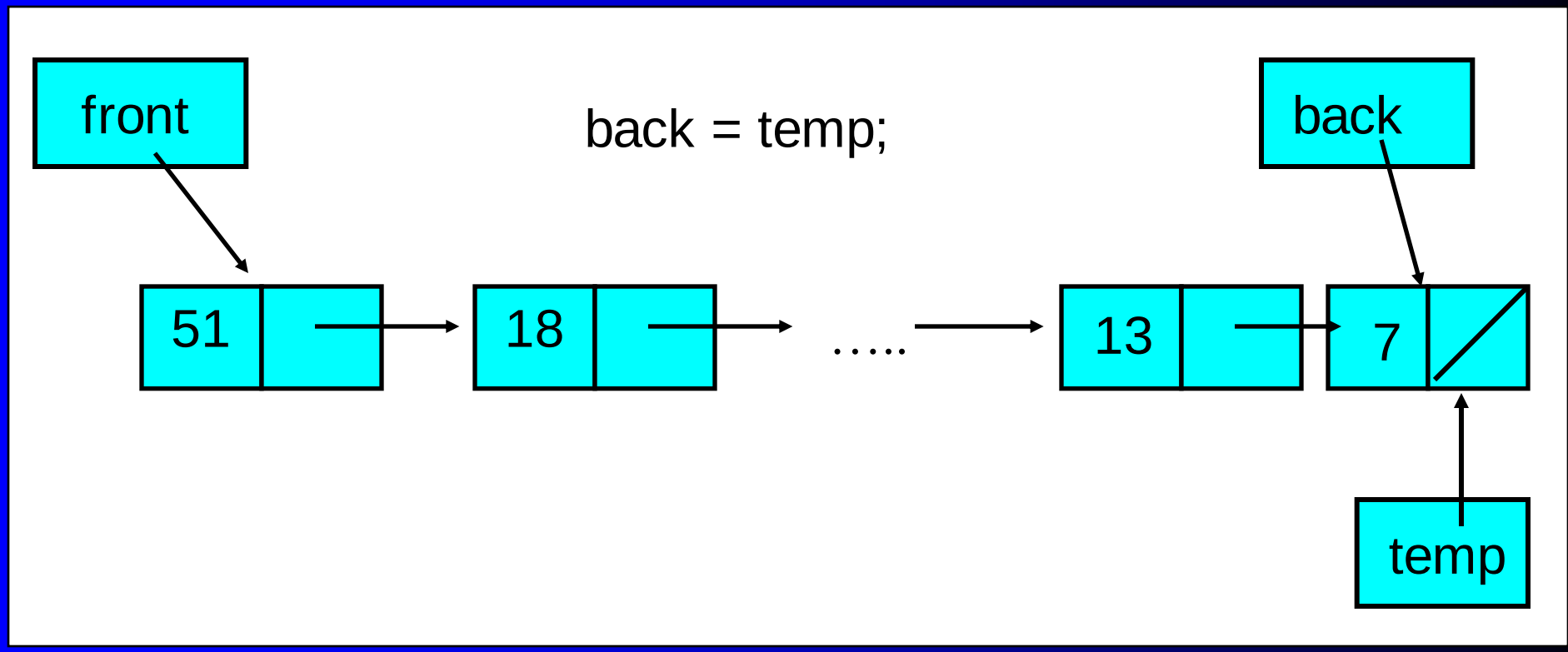
# Linked list implementation - insert



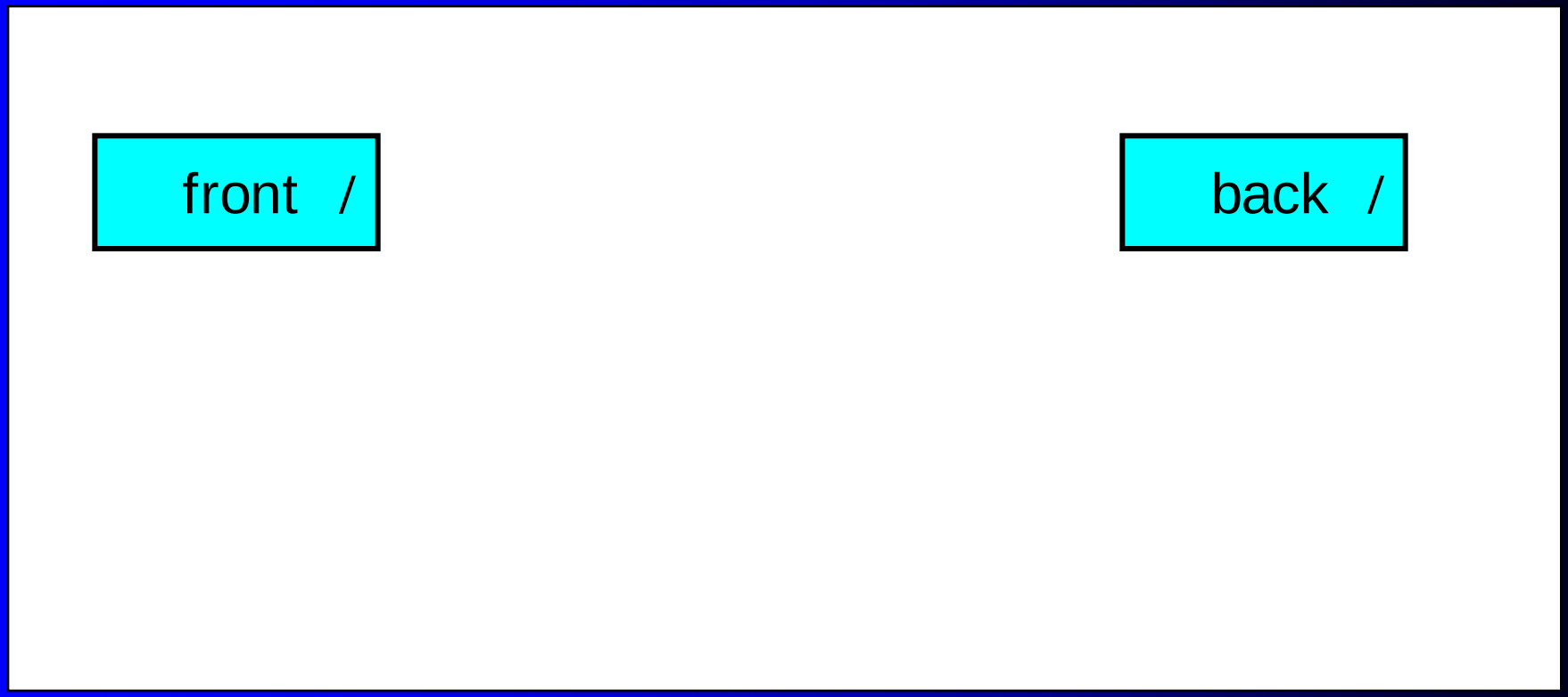
# Linked list implementation - insert



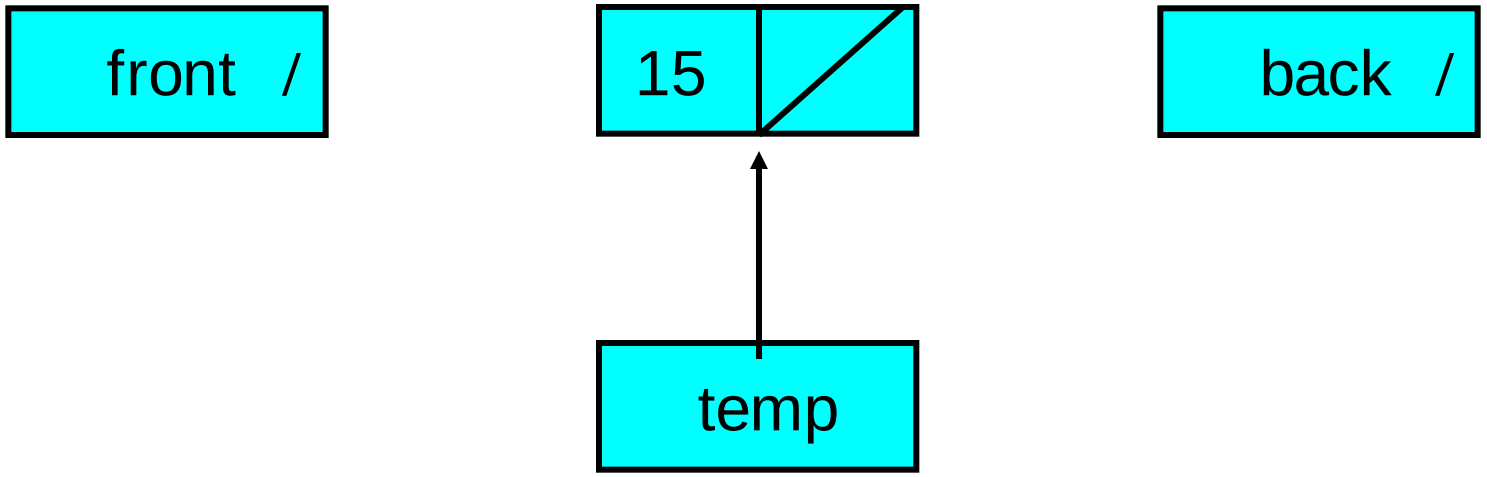
# Linked list implementation - insert



# Insert to an empty queue

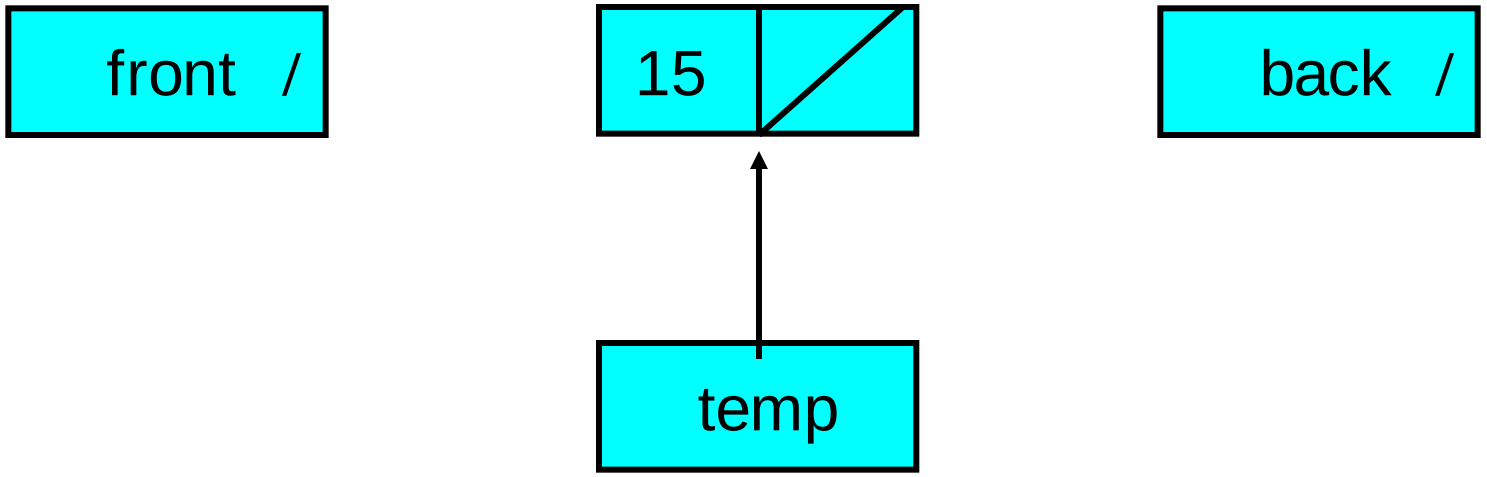


# Insert to an empty queue

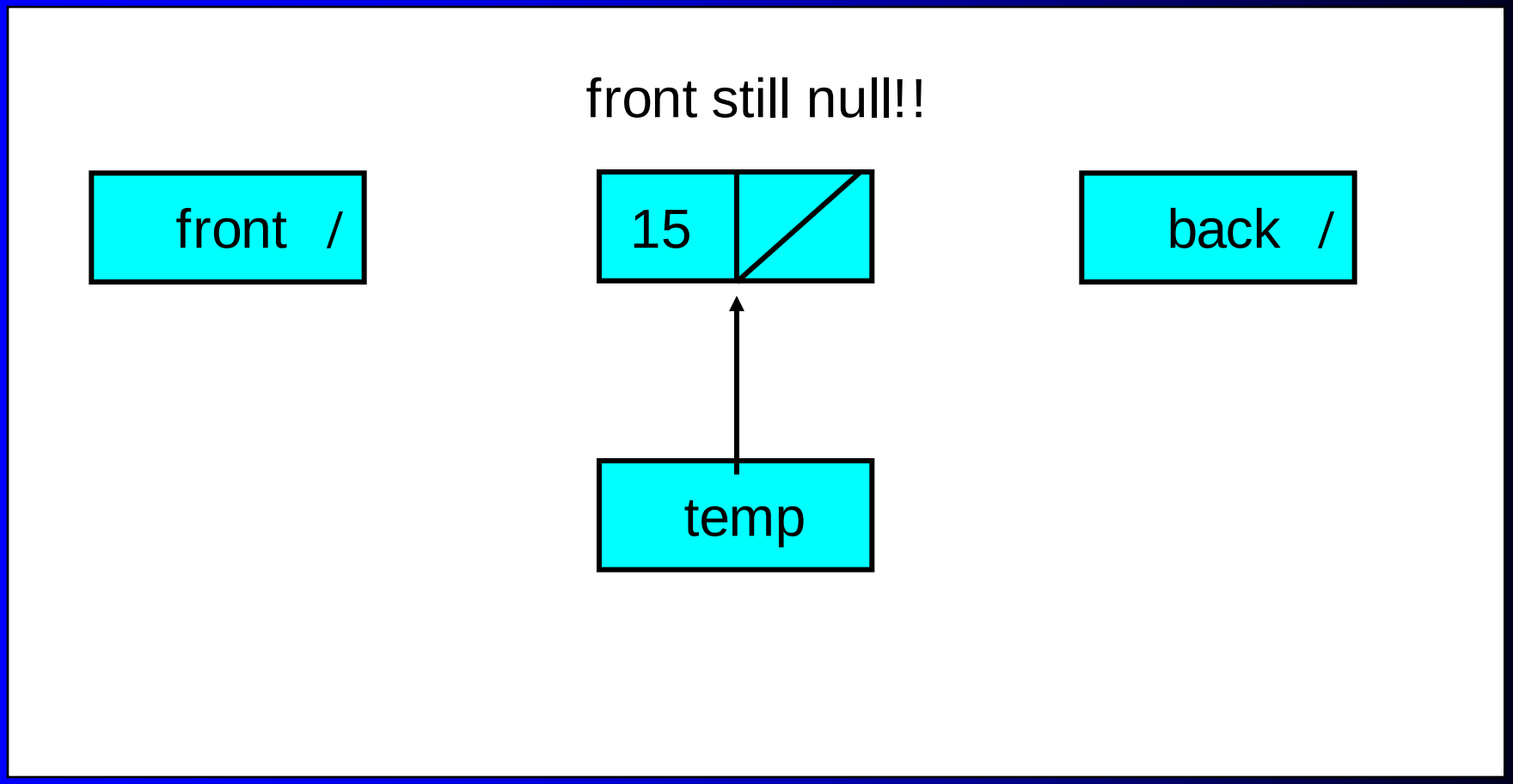


# Insert to an empty queue

back.next = temp ??



# Insert to an empty queue



# Special case

```
if (front == null) {
   front = temp;
   back = temp;
}
else
   back.next = temp;
```

# Linked list implementation

- insert

```
public void insert (Object new_data) {
   QueueNode temp = new QueueNode(new_data);

   if (isEmpty())
            front = temp;
   else
            back.next = temp;

   back = temp;
}
```

# Linked list implementation

- get

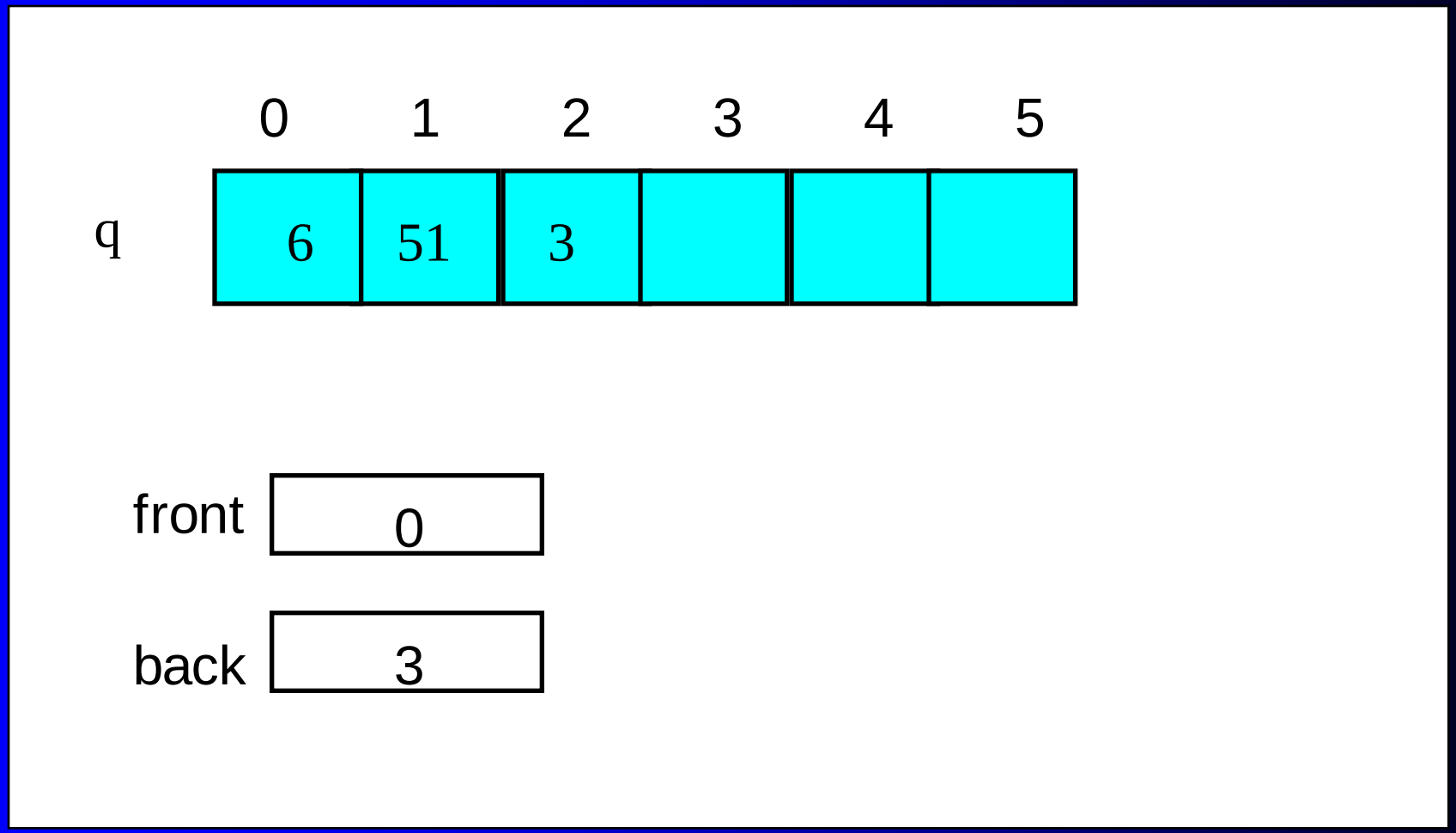
```
public Object get () {
   if (isEmpty() ) {… error …}

   Object result = front.data;

   front = front.next;
   if (front == null)
      back = null;

   return result;
}
```

# Class exercise

- Write code for isEmpty

```
public boolean isEmpty () {
    ….
}
```

# Array implementation of a queue



# Insertion

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| q | 6 | 51 | 3 | | | |

| | |
|---|---|
| front | 0 |
| back | 3 |

To insert a new cell:

```
q[back] = new_cell;
back++;
```

# Insertion

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| q | 6 | 51 | 3 | 9 | | |

| | |
|---|---|
| front | 0 |
| back | 3 |

To insert a new cell:

```
q[back] = new_cell;
back++;
```

# Insertion

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| q | 6 | 51 | 3 | 9 | | |

| | |
|---|---|
| front | 0 |
| back | 4 |

To insert a new cell:

q[back] = new_cell;

back++;

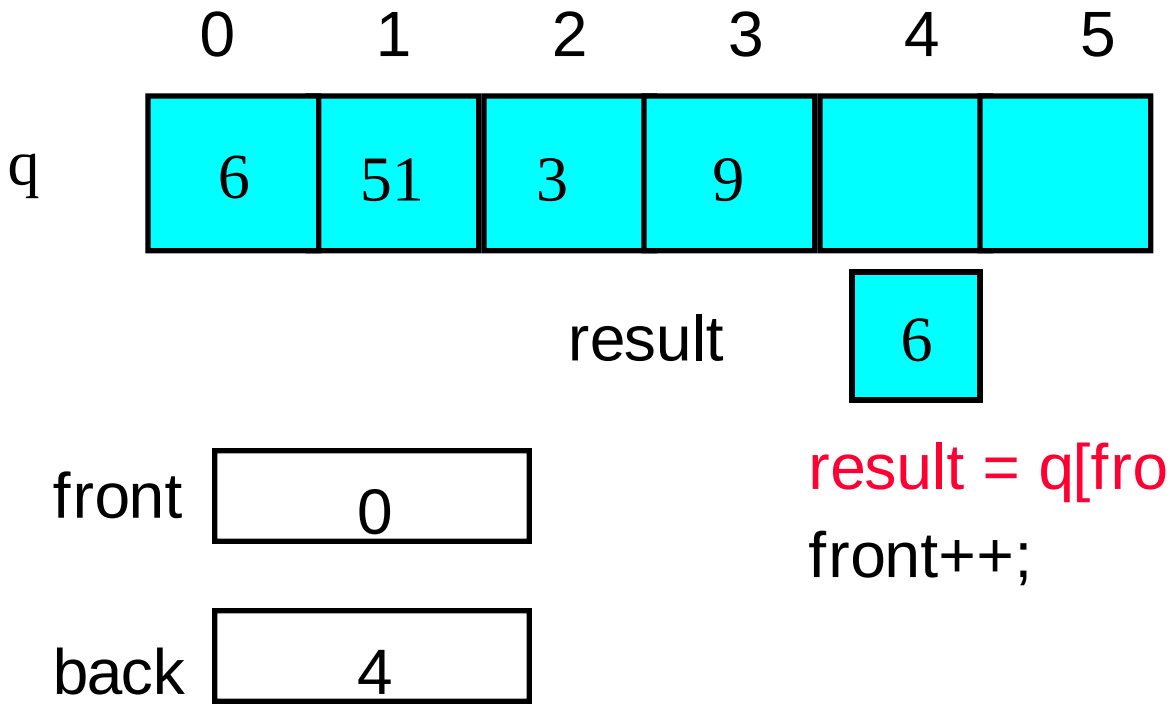
# Getting an item from the queue

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| q | 6 | 51 | 3 | 9 | | |

| | |
|---|---|
| front | 0 |
| back | 4 |

```
result = q[front];
front++;
```

# Getting an item from the queue




result = q[front];

front++;

# Getting an item from the queue

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| q | | 51 | 3 | 9 | | |

| | |
|---|---|
| front | 1 |
| back | 4 |

result = q[front];

front++;

# Problem: insert 34 below

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| q | | | 3 | 9 | 14 | 23 |

| | |
|---|---|
| front | 2 |
| back | 6 |

# Solution: wrap-around list

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| q | 34 | | 3 | 9 | 14 | 23 |

| | |
|---|---|
| front | 2 |
| back | 1 |

# Solution: wrap-around list

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| q | 34 | | 3 | 9 | 14 | 23 |

| | |
|---|---|
| front | 2 |
| back | 1 |

```
if (back == 6) back = 0;
q[back] = new_cell;
back++;
```

# JAVA declaration & initialization

```
public class Queue {
    public final static int MAX_QUEUE_SIZE = 50;
    private Object[] storage;
    private int front = -1;
    private int back = 0;

    public Queue() {
            storage = new Object[MAX_QUEUE_SIZE]; }
    public void insert(Object x) { ... ...}
    public Object get() {... ...}
    public boolean isEmpty() {... ...}
    public boolean isFull() {... ...}
}
```

# Queue operations

- insert

```
public void insert (Object x) {
    if (isFull()) { ... queue full ... }

    storage[back] = x;

    if(isEmpty())
        front = back;

    back = (back + 1) % MAX_QUEUE_SIZE;
}
```

# Queue operations

- get

```
public Object get () {
    if (isEmpty()) { ... queue empty ... }

    Object result = storage[front];
    front = (front + 1) % MAX_QUEUE_SIZE;

    if (front == back) {
            front = -1;
            back = 0;
    }
    return result;
}
```

# Queue operations

- isEmpty

```
public boolean isEmpty () {
    return front == -1;
}
```

# Queue operations

- isFull

```
public boolean isFull () {
    return back == front;
}
```

# Use of the Queue ADT

- Simulations of real-life queues (e.g., banks, supermarkets, traffic lights, etc)
- Program design independent of implementation