CS-202

Dynamic Data Structures (Pt.2 & Pt.3)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday
			Lab (9:00-12:50)	
	CLASS		CLASS	
PASS Session	PASS Session	Project DEADLINE	**Project DEADLINE**	

Your 8th Project Deadline is *shifted* from Wednesday 11/15 to Thursday 11/16.

- PASS Sessions held Monday-Tuesday, get all the help you may need!
- 24-hrs delay after Project Deadline incurs 20% grade penalty.
- Past that, NO Project accepted. Better send what you have in time!

Today's Topics

Dynamic Data Structures

Queues(s)

- Array-based
- Node-based

Queue(s)

The Basics

A Dynamic Data Structure type, with its own semantics.

An ordered group of homogeneous items.

- Has two ends, a *front* and a *back*.

Operational semantics:

- Elements are added at the *back* (rear).
- Elements are removed from the *front* (start).
- Middle elements are inaccessible.

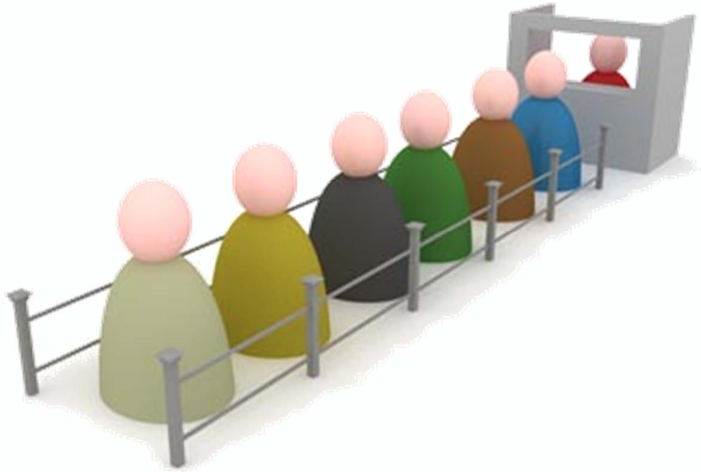
Queue(s)

The Basics

A Dynamic Data Structure type, with its own semantics.

Operational semantics (continued):

- First-In, First-Out (FIFO) property.
- The first element added first, is the first to be removed.



Applications

Queues are appropriate to handle for many real-world situations:

- Example: Waiting lists.
  In bureaucracy - A line to be served at the DMV.

Queues have numerous computer (science)-related applications:

- Example: Access to shared resources.
- For a CPU – Multiprogramming.
  For a printer – Serving a request to print a document.

Cross-field simulations & case-studies:

- Strategies to reduce the wait involved in an application.

https://coe.neu.edu/healthcare/pdfs/publications/intro_computer_simulation_healthcare_case_study.pdf

Queue Operations

A complete Queue-based ADT implementation has to support the following functionalities:

- Creation of an empty Queue.
- Destroying a Queue.
- Determining whether a Queue is empty.
- Adding (at the back) a new element to the Queue.
- Removal (from the front) of the item that was added earliest.
- Retrieval of the earliest added the item (at the front).

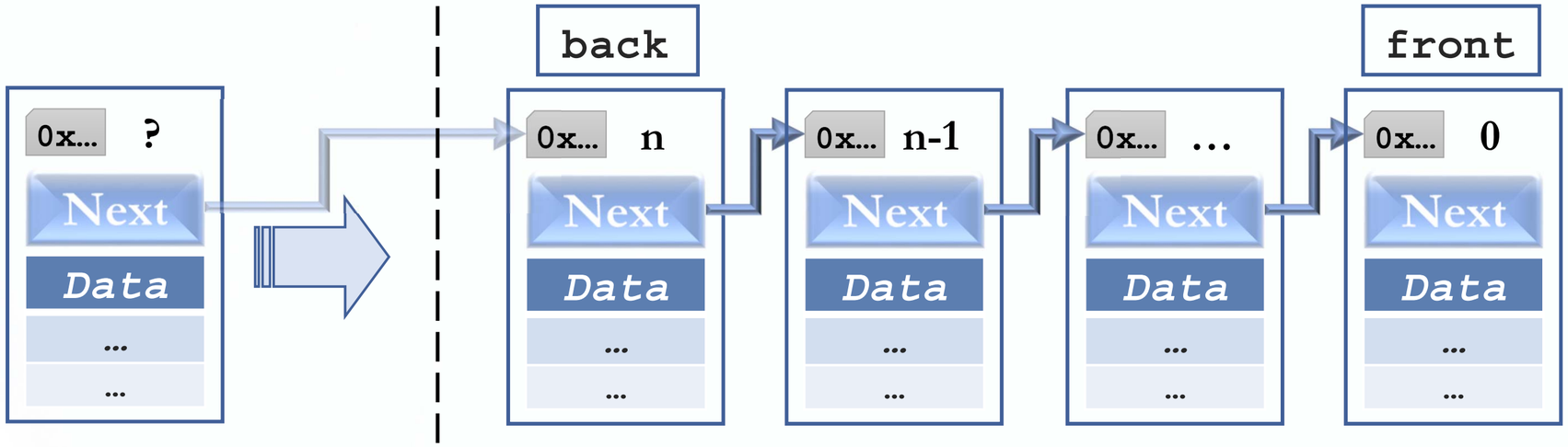
Queue Operations

Queue `push()`-ing

When a new element needs to be added to the queue:

- New people enter at the end of the line.
- New service requests made to a server.

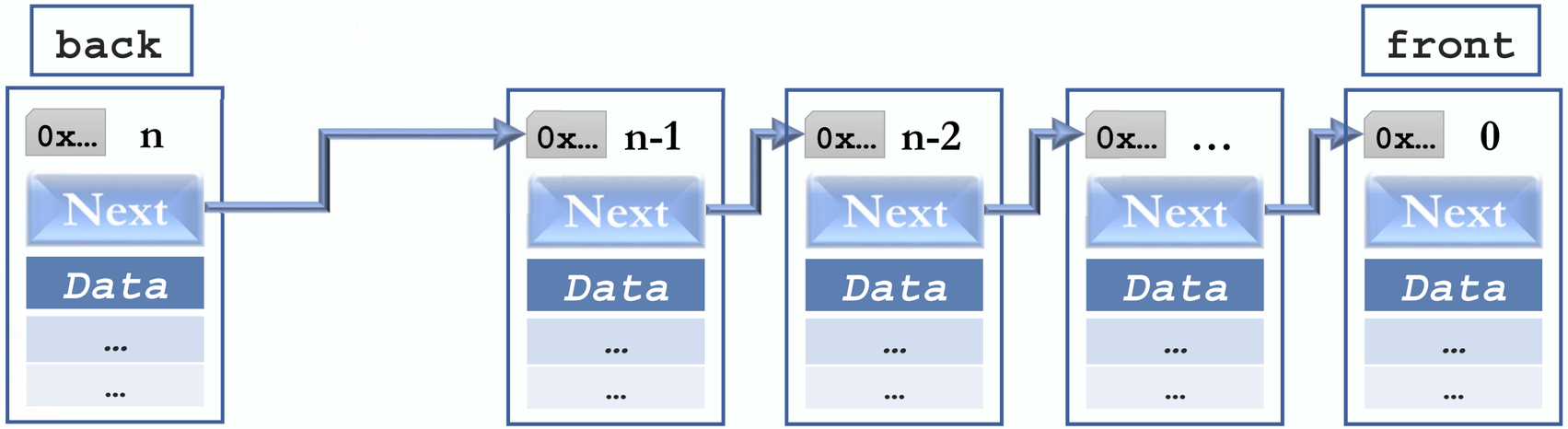
Called an "**`enqueue`**" operation (also **`push`**, **`addElement`**, etc.)



Queue Operations

Queue `push()`-ing

```
void push(const DataType& value);
// Inserts an element at the back of a queue.
// Precondition: value is the value of the element to be
inserted.
// Postcondition: If the insertion is successful, a new
DataType element of value is at the back of the queue.
```



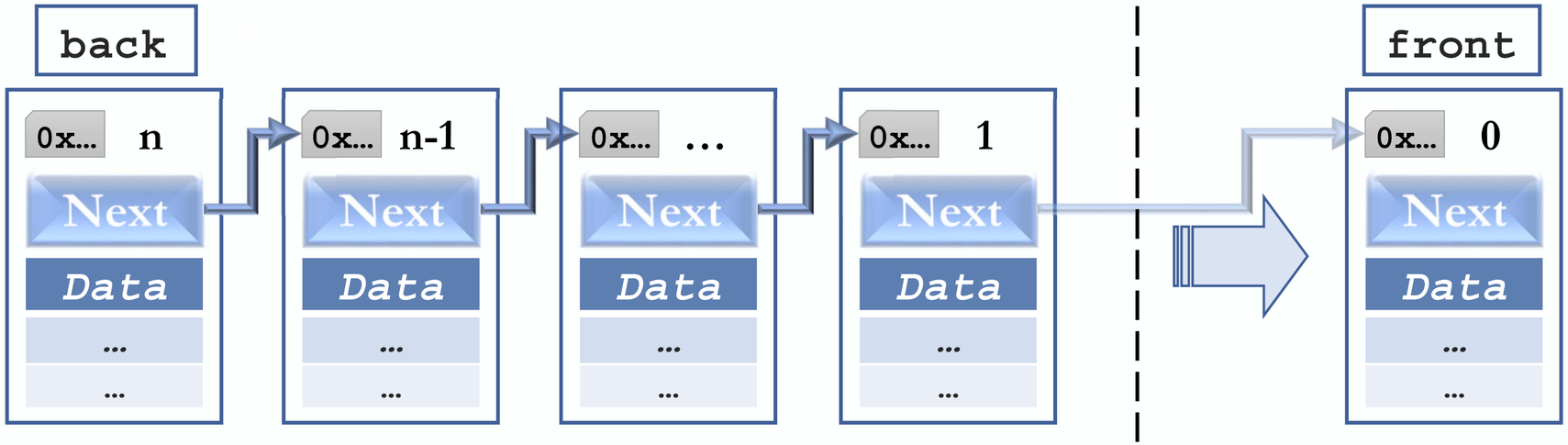
Queue Operations

Queue `pop()`-ping

When an element needs to be removed from the queue:

- Front-of-line person goes away.
- A service request has been completed.

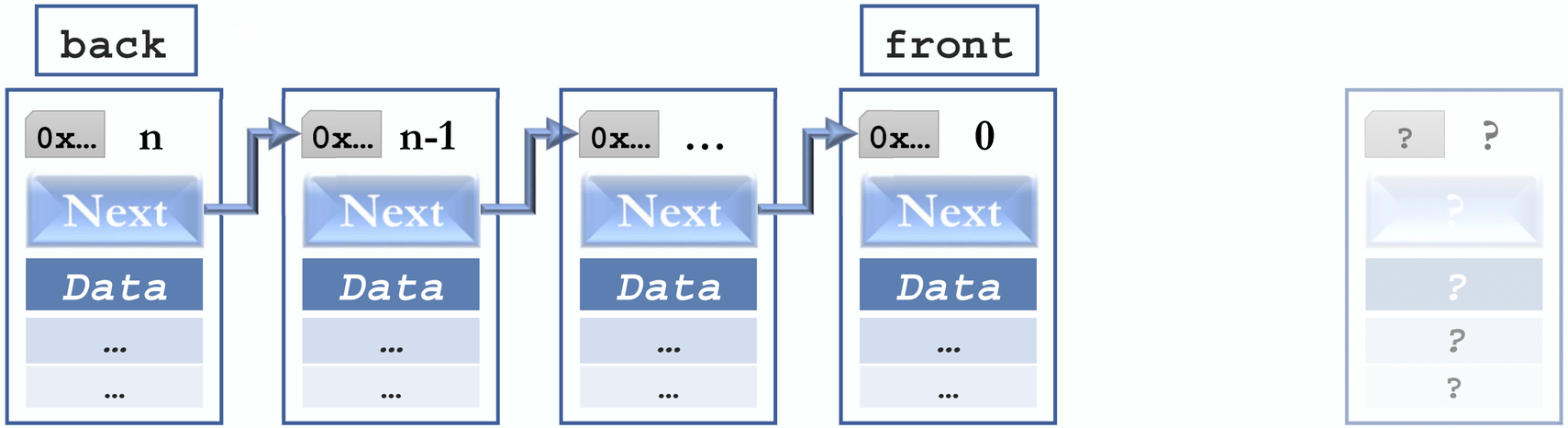
Called an "**`dequeue`**" operation (also **`pop`**, **`removeElement`**, etc.)



Queue Operations

Queue `pop()`-ing

```
void pop();
// Dequeues the front of a queue.
// Precondition: None.
// Postcondition: If the queue is not empty, the element that
was added to the queue earliest is deleted.
```



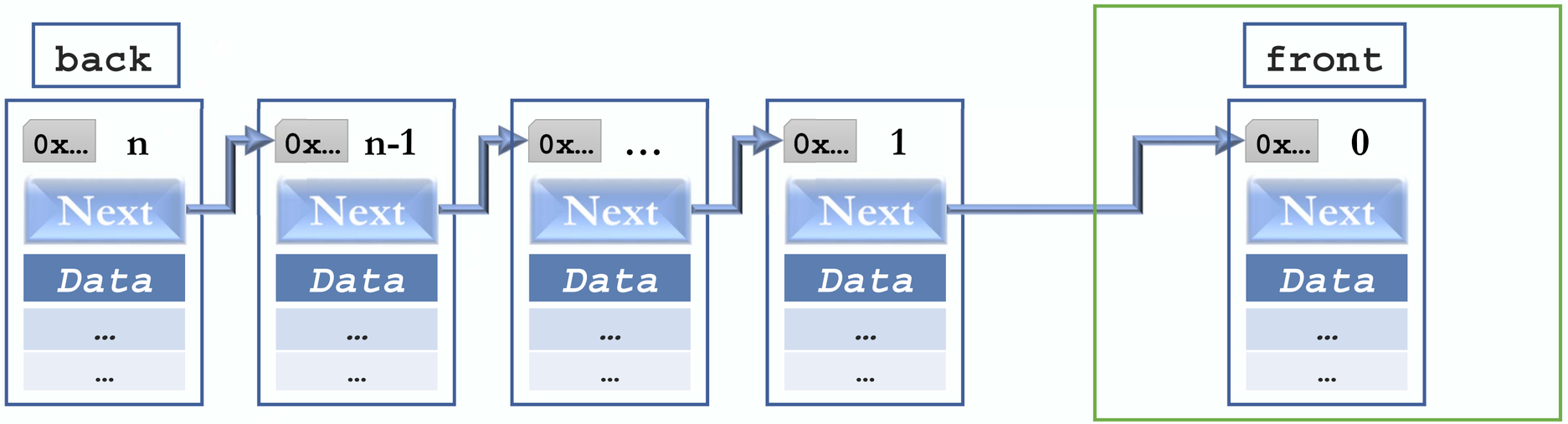
Queue Operations

Queue `front()`

When the front element needs to be accessed:

- Get front-of-line person to teller.
- Acquire service request to forward for execution.

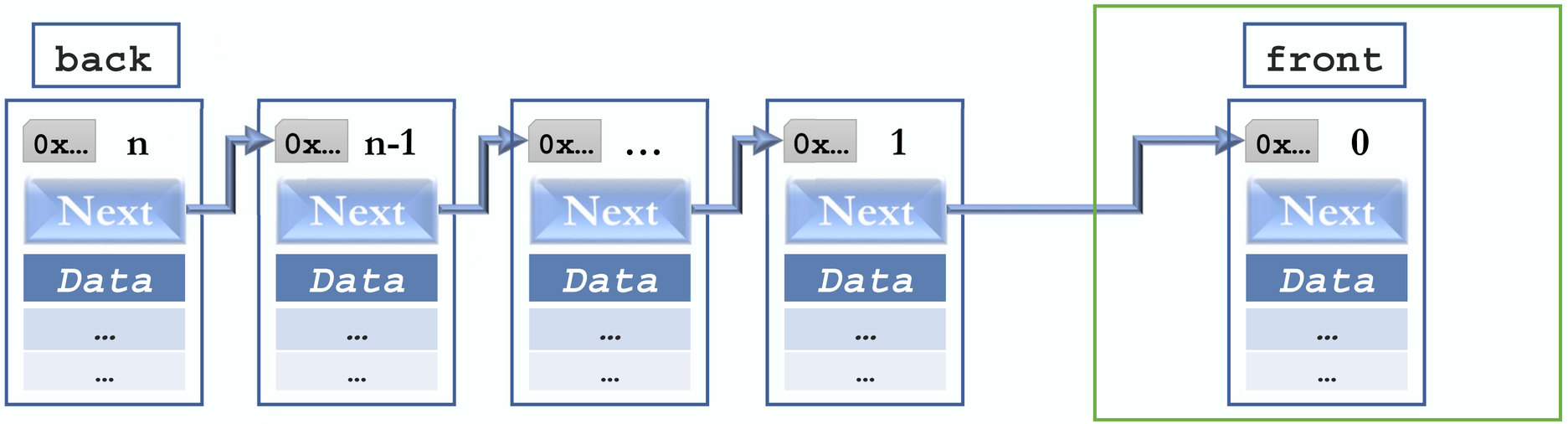
Called an "**`getFront`**" operation (also **`frontElement`**, etc.)



Queue Operations

Queue front()

```
DataType& front();
const DataType& front() const;
// Retrieves the element at the front of a queue
// Precondition: The queue is not empty.
// Postcondition: If the queue is not empty, the return value
is a (const) reference to the earliest added element.
```

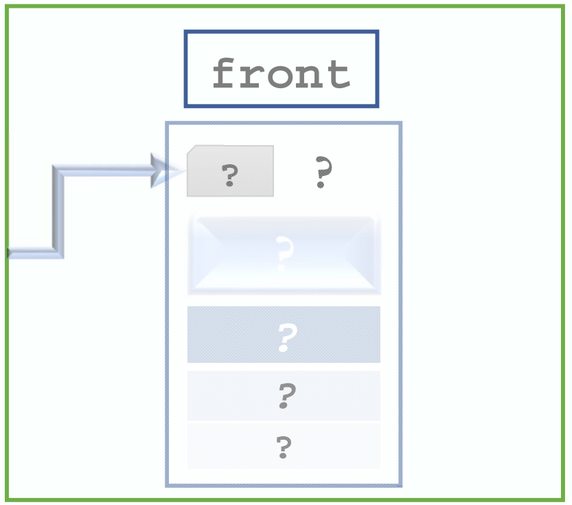


Queue Operations

Queue `front()`

```
DataType& front();
const DataType& front() const;
// Retrieves the element at the front of a queue
// Precondition: The queue is not empty.
// Postcondition: If the queue is not empty, the return value
is a (const) reference to the earliest added element.
```

- Have to return a valid Object reference.
- Have to first check that the queue is not empty!

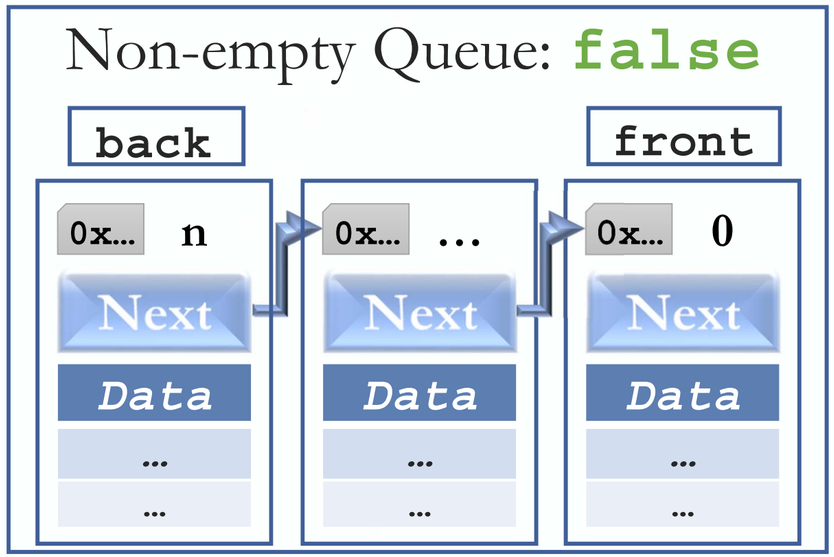


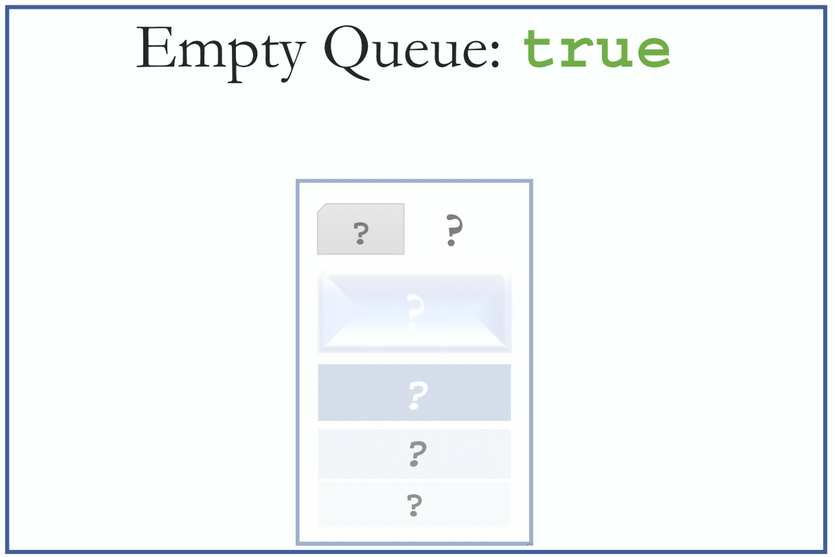
Remember: From the C++11 standard:
[dcl.ref] [...] a `NULL` reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the "object" obtained by dereferencing a `NULL` pointer, which causes undefined behavior.

Queue Operations

Queue `empty()`

```
bool empty() const;
// Determines whether the queue is empty.
// Precondition: None.
// Postcondition: Returns true if the queue is empty;
otherwise returns false.
```





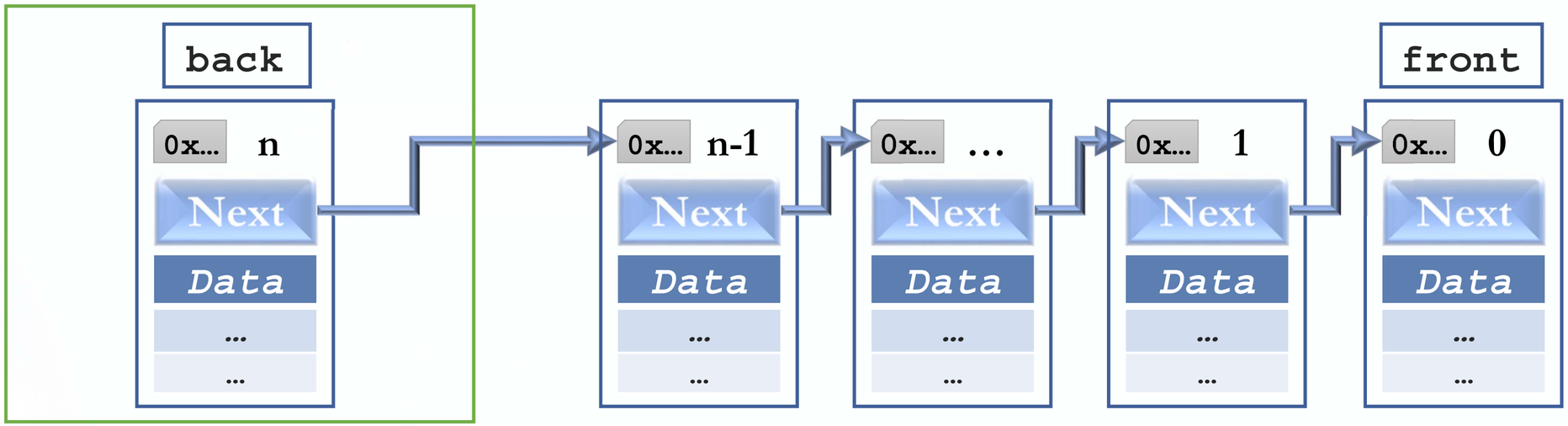
Queue Operations

Queue `back()` (outside of specifications)

When the last element needs to be accessed:

- Get last-in-line person's details.
- Peek at the expected load of the last-in-line service request.

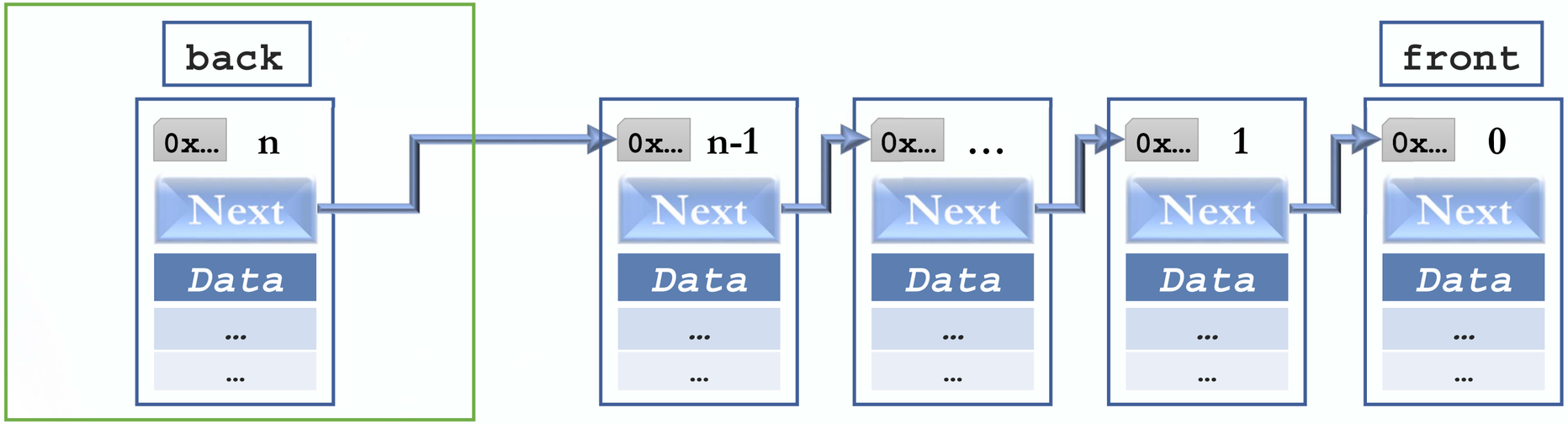
Called an "**`getBack`**" operation (also **`back`**, **`lastElement`**, etc.)



Queue Operations

Queue `back()` (outside of specifications)

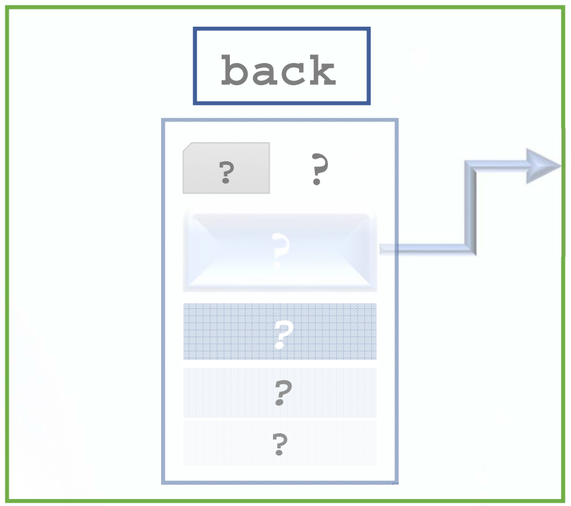
```
DataType& back();
const DataType& back() const;
// Retrieves the element at the end of a queue
// Precondition: The queue is not empty.
// Postcondition: If the queue is not empty, the return value
is a (const) reference to the earliest added element.
```



Queue Operations

Queue `back()` (outside of specifications)

```
DataType& back();
const DataType& back() const;
// Retrieves the element at the end of a queue
// Precondition: The queue is not empty.
// Postcondition: If the queue is not empty, the return value
is a (const) reference to the last added element.
```



- Have to return a valid Object reference.
- Have to first check that the queue is not empty!

Remember: From the C++11 standard:
[dcl.ref] [...] a `NULL` reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the "object" obtained by dereferencing a `NULL` pointer, which causes undefined behavior.

Queue Implementations

“Standard” Implementations

A complete Queue-based ADT implementation encompasses a subset of List ADT functionalities.

- Exploit List ADT approaches.
- An Array-based implementation.
- A Pointer (Node)-based implementation.

  A linear linked list with two external references:

  - A reference to the front element.
  - A reference to the back element.

Array-based Queue(s)

Array-based Implementation(s)

A Queue can be implemented with an array, as shown here.

- An array of `int`s to hold an represent a Queue of `int`s.
- This Queue contains the integers 4 (at the front), 8 and 6 (at the rear).
- We do not care about any elements other than those three.

The "valid" array elements subset.

These array elements do not concern the program at this point.

m_arr [0]	m_arr [1]	m_arr [2]	m_arr [3]	m_arr […]	m_arr [98]	m_arr [99]
int	int	int	int	int	int	int
4	8	6	…	…	…	…

Array-based Queue(s)

Array-based Implementation(s)

A Queue can be implemented with an array, as shown here.

The "easiest" implementation keeps track of:

- The number of elements in the Queue.
- The index of the front (first) element.
- The index of the back (last) element.

And "remembers":

- The underlying container's (the array's) total size.

```
m_size   := 3
m_front  := 0
m_back   := 2
m_maxsize := 100
```

m_arr [0]	m_arr [1]	m_arr [2]	m_arr [3]	m_arr […]	m_arr [98]	m_arr [99]
int	*int*	*int*	*int*	*int*	*int*	*int*
4	*8*	*6*	…	…	…	…

Array-based Queue(s)

Array-based Implementation(s)

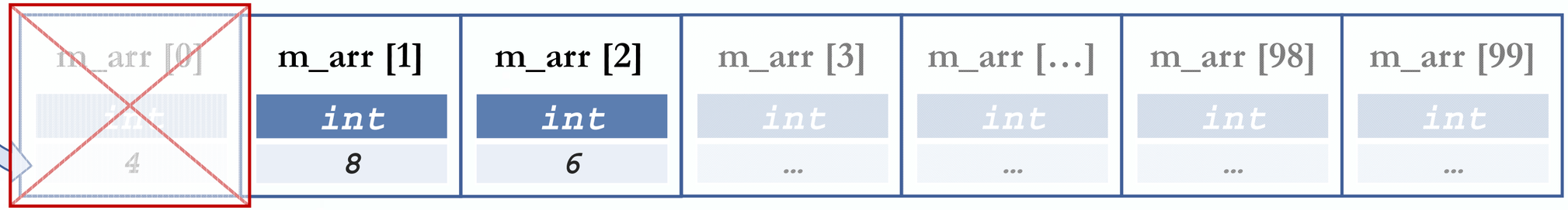
A Queue `pop()` (**`dequeue`**) operation – *Naïve* approach.

When an element is removed from the Queue:

- The size is decremented.
- The front is changed.

```
m_size  := 2
m_front := 1
m_back  := 2
m_maxsize := 100
```

Note:
`pop()` does not *clear* contents, it only updates the Queue values that keep track of its state.



Array-based Queue(s)

Array-based Implementation(s)

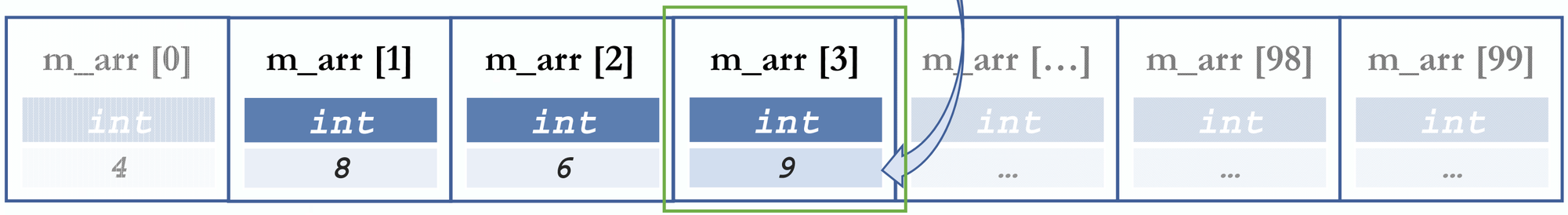
A Queue **`push()`** (**`enqueue`**) operation – *Naïve* approach.

When an element is pushed to the Queue:

- The size is incremented.
- The back is changed.

```
m_size  := 3
m_front := 1
m_back  := 3
m_maxsize := 100
```

Note:
`push(...)` *overwrites* contents, and also updates the Queue values that keep track of its state.

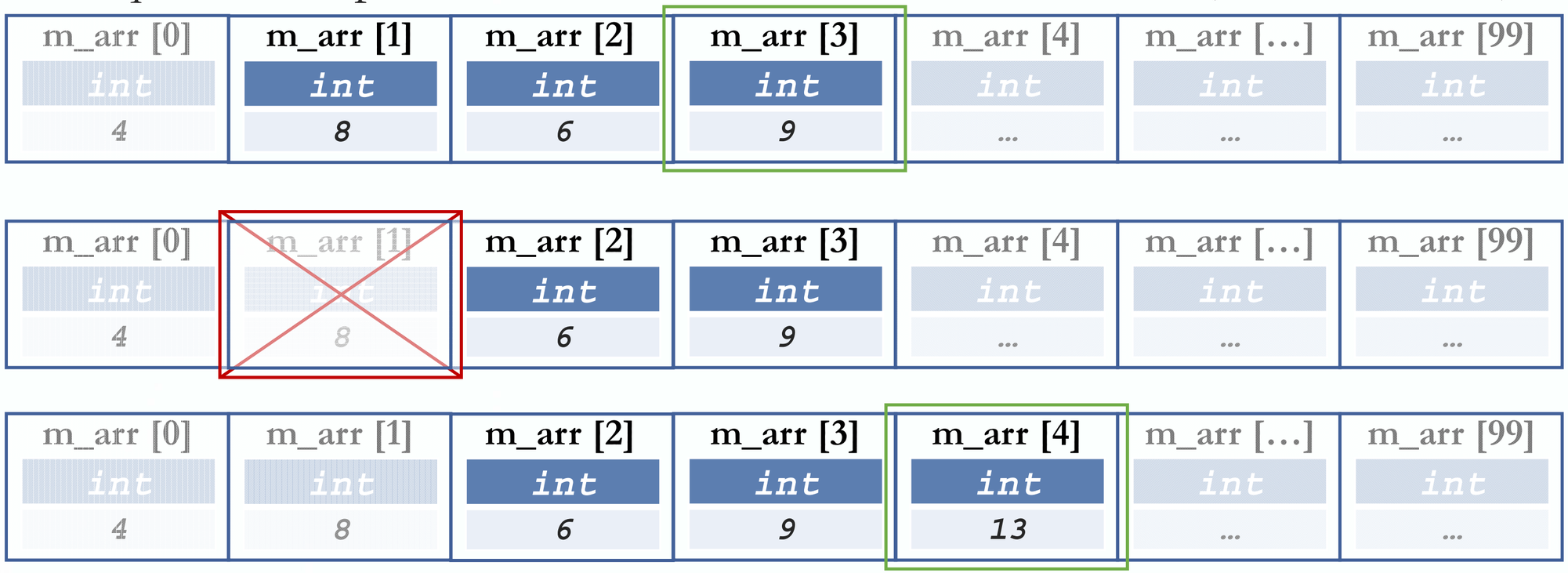


Array-based Queue(s)

Array-based Implementation(s)

Queue *Naïve* approach issues.

For a sequence of operations: ADADADADADADADADA... (A:Add, D: Delete)

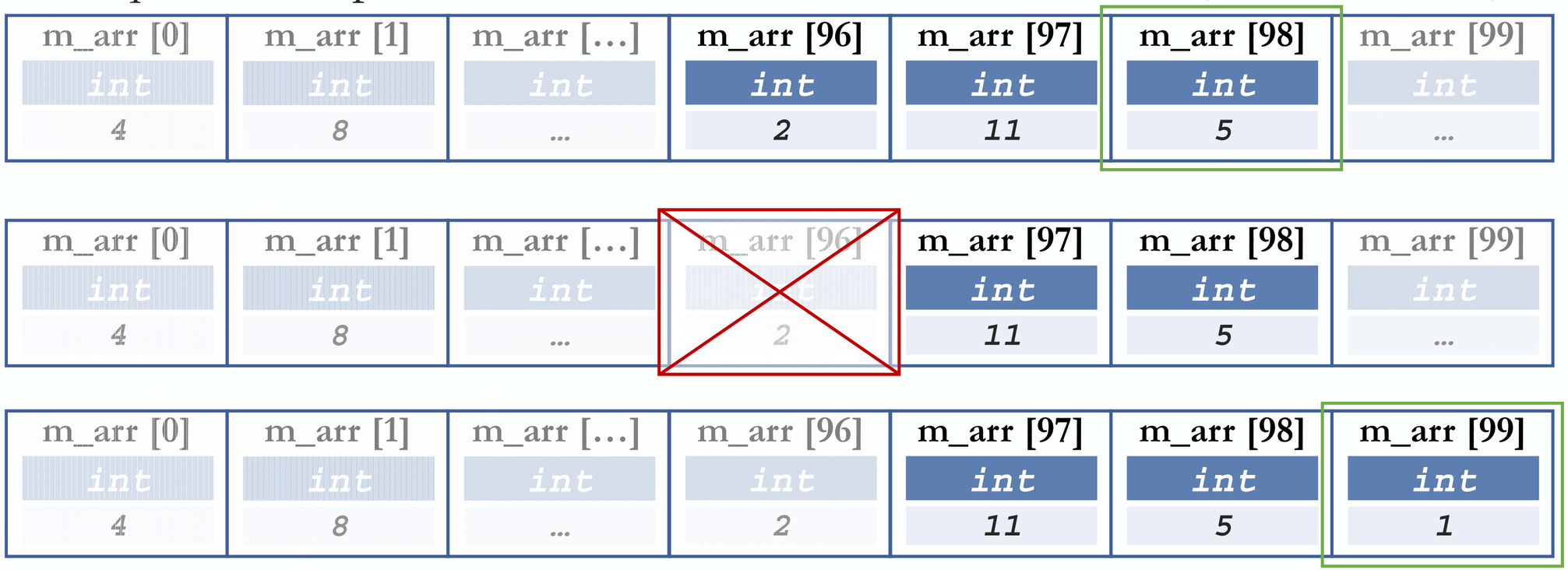


Array-based Queue(s)

Array-based Implementation(s)

Queue *Naïve* approach issues.

For a sequence of operations: ADADADADADADADADA... (A:Add, D: Delete)



Array-based Queue(s)

Array-based Implementation(s)

Queue *Naïve* approach issues.

- Eventually **`m_back`** index points to last array position **`m_maxsize-1`**.
- Looks like the underlying array space is up (can't **`push(…)`** more elements).
- In reality: Queue only has two or three elements, array is empty in front.

```
m_size   := 3
m_front  := 97
m_back   := 99
m_maxsize := 100
```

m_arr [0]	m_arr [1]	m_arr […]	m_arr [96]	m_arr [97]	m_arr [98]	m_arr [99]
int	int	int	int	int	int	int
4	8	…	2	11	5	1

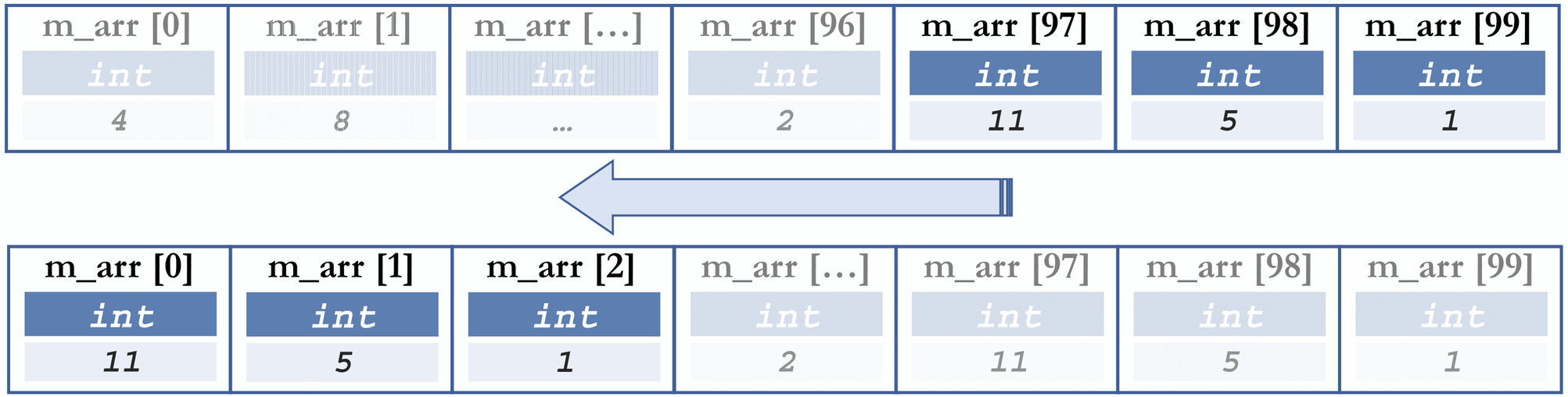
???

Array-based Queue(s)

Array-based Implementation(s)

A "simple" solution – Upon condition of Queue rear overflow:

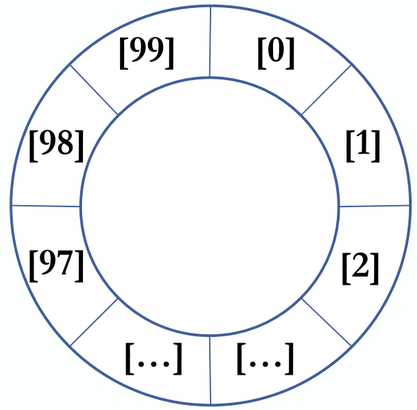
- Check value of front, and if there is room,
- Slide all queue elements toward first array position.
- Works best with small Queue sizes.



Array-based Queue(s)

Array-based Implementation(s)

An "elegant" solution – The circular array paradigm:



m_arr [0]	m_arr [1]	m_arr […]	m_arr [96]	m_arr [97]	m_arr [98]	m_arr [99]
char	char	char	char	char	char	char
…	…	…	…	A	B	C

```
charQueue.push('D');
```

???

```
m_size := 3          m_front := 97
m_maxsize := 99      m_back := 99
```

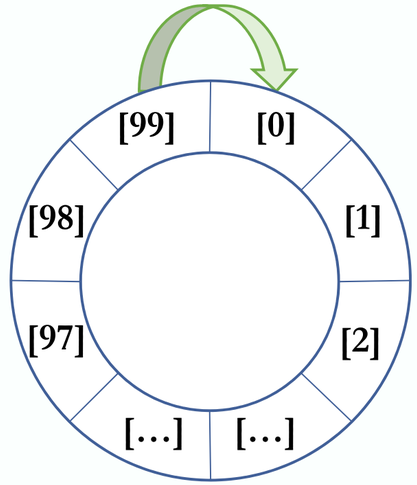
Advance m_back to next circular array position !

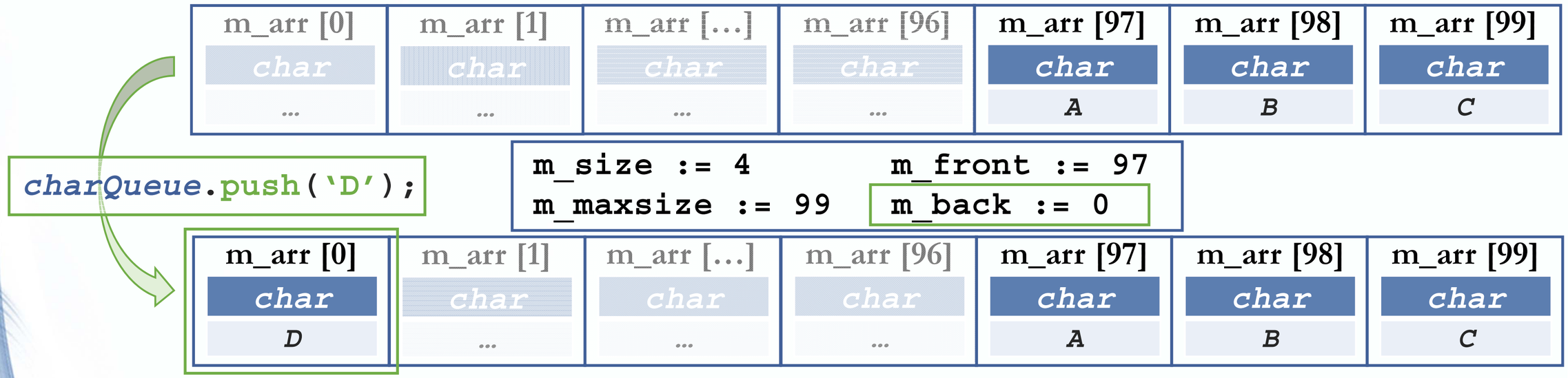
```
m_back = (m_back + 1) % m_maxsize;
```

Array-based Queue(s)

Array-based Implementation(s)

An "elegant" solution – The circular array paradigm:





Array-based Queue(s)

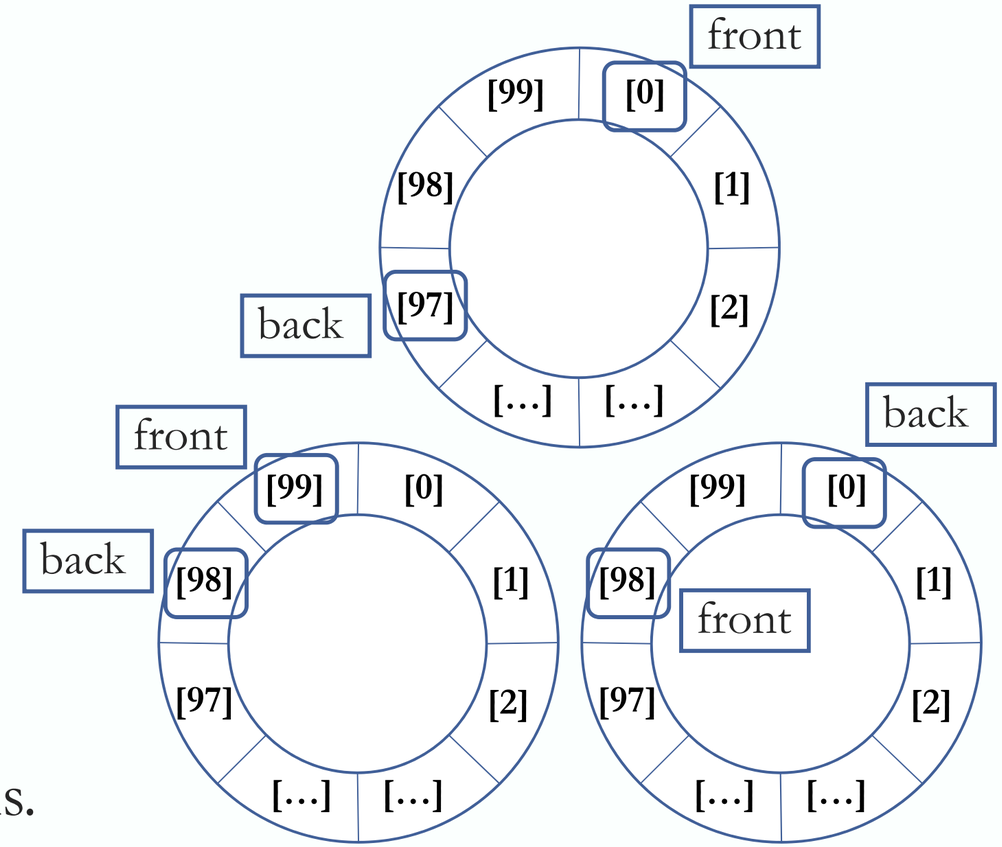
Array-based Implementation(s)

The circular array:

- Eliminates issue of rightward drift.

But:

- Values of **`m_front`** and **`m_back`** can no longer directly distinguish between full-Queue and empty-Queue-empty conditions.

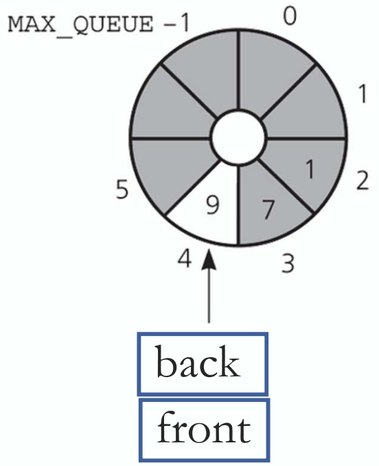


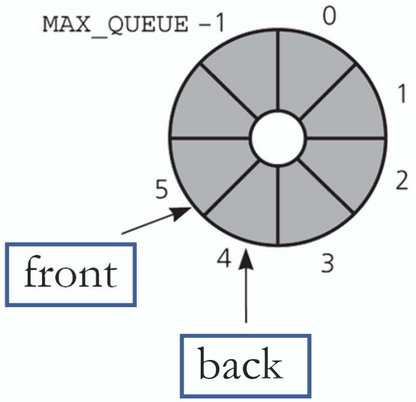
Array-based Queue(s)

Array-based Implementation(s)

a) *front* passes *back*
when the queue becomes empty.

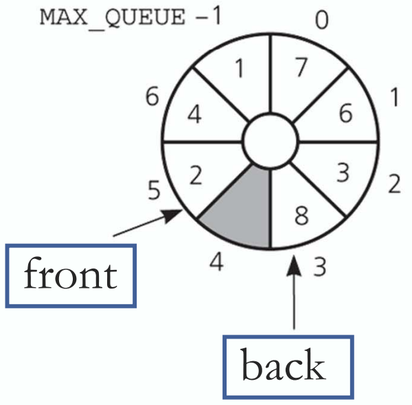
- Queue with single element:
  `pop()` → Queue becomes empty.

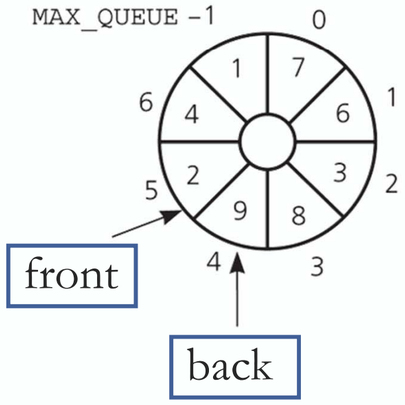




b) *back* catches up to *front*
when the queue becomes full.

- Queue with single empty slot:
  `push(9)` → Queue becomes full.





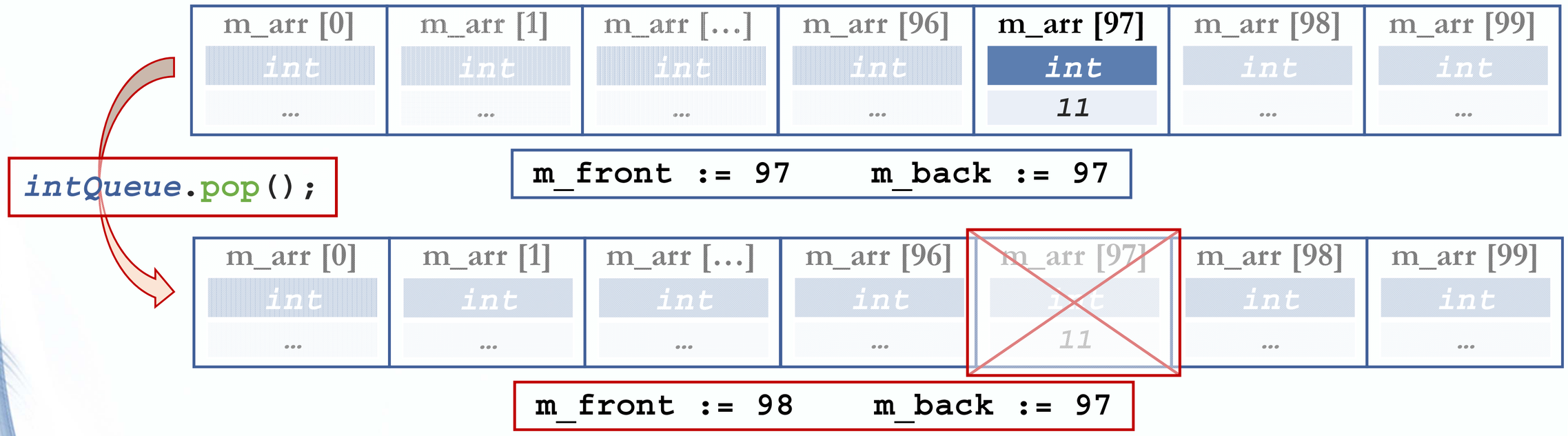
Array-based Queue(s)

Array-based Implementation(s)

Circular array issues (continued):

- Cases with identical *front* & *back* index values.

  An empty Queue, **`dequeue`** operation:



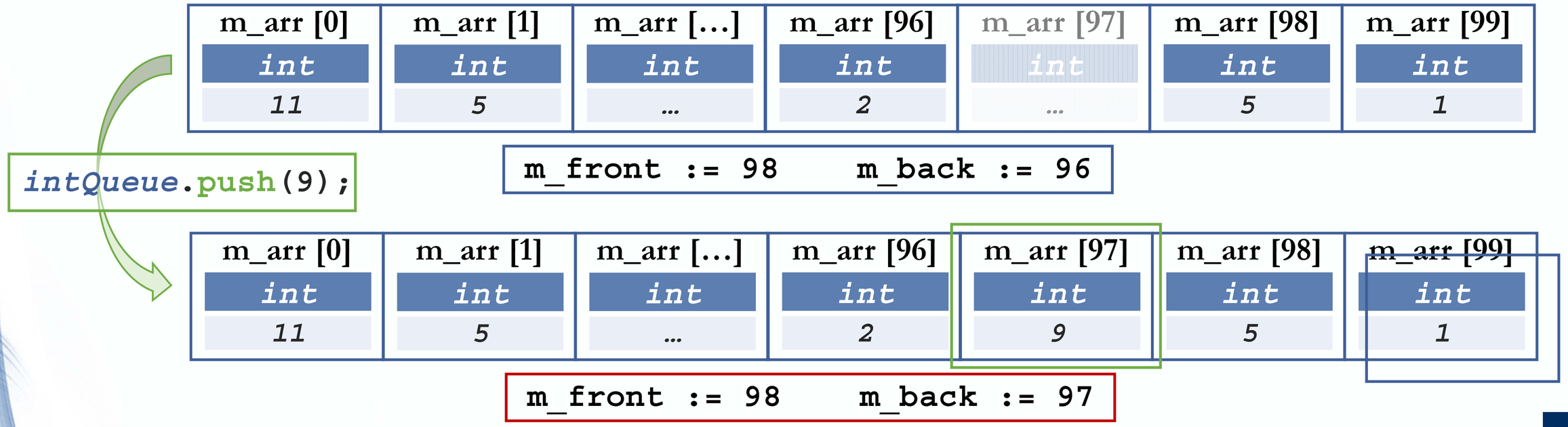
Array-based Queue(s)

Array-based Implementation(s)

Circular array issues (continued):

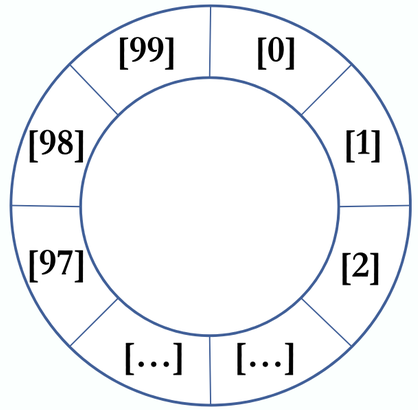
- Cases with identical *front* & *back* index values.

  A full Queue, **`enqueue`** operation:



Array-based Queue(s)

Array-based Implementation(s)



Circular array specifications to detect full-Queue & empty-Queue conditions:

- Keep a count of the queue elements (**`m_size`**).
- Incremented when new element **`push`**'ed.
- Decremented when element **`pop`**'ped.

Queue Initialization:

- Set **`m_front`** to 0.
- Set **`m_back`** to **`m_maxsize-1`**.
- Set **`m_size`** to 0.

Array-based Queue(s)

Array-based Implementation(s)

Queue Insertion (at the *back*):

```
m_back = (m_back+1) % m_maxsize;
m_arr[m_back] = newElement;
++m_size;
```

Queue Removal (from the *front*):

```
m_front = (m_front+1) % m_maxsize;
--m_size;
```

Keeping track of Queue size via a helper element-counting variable.

Advancing *back* & *front* indexes in the array as data are `push`'ed & `pop`'ped.

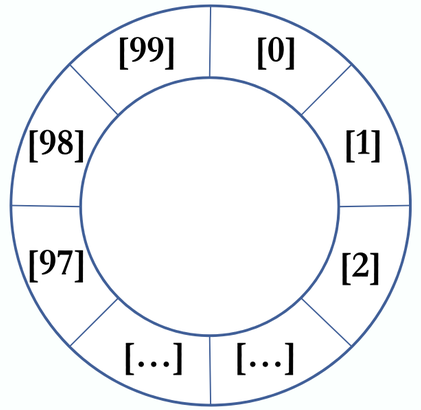
Array-based Queue(s)

Array-based Implementation(s)

```
typedef pod-or-class-or-struct-type DataType;
class Queue{
  public:
    Queue();
    Queue(int count, const DataType& val);
    Queue(const Queue& other);
    ~Queue();
    Queue& operator=(const Queue& other);
    bool empty() const;
    int size() const;
    void push(const DataType& value);
    void pop();
    void clear();
    DataType& front();
    DataType& back();
  private:
    DataType *m_arr;
    int       m_front, m_back;
    int       m_size, m_maxsize;
};
```

Array-based Queue(s)

Array-based Implementation(s)



Remember:

Detecting full-Queue & empty-Queue conditions:

- Keep a count of the queue elements (**`m_size`**).
- Incremented when new element **`push`**'ed.
- Decremented when element **`pop`**'ped.

Array-based Queue variations:

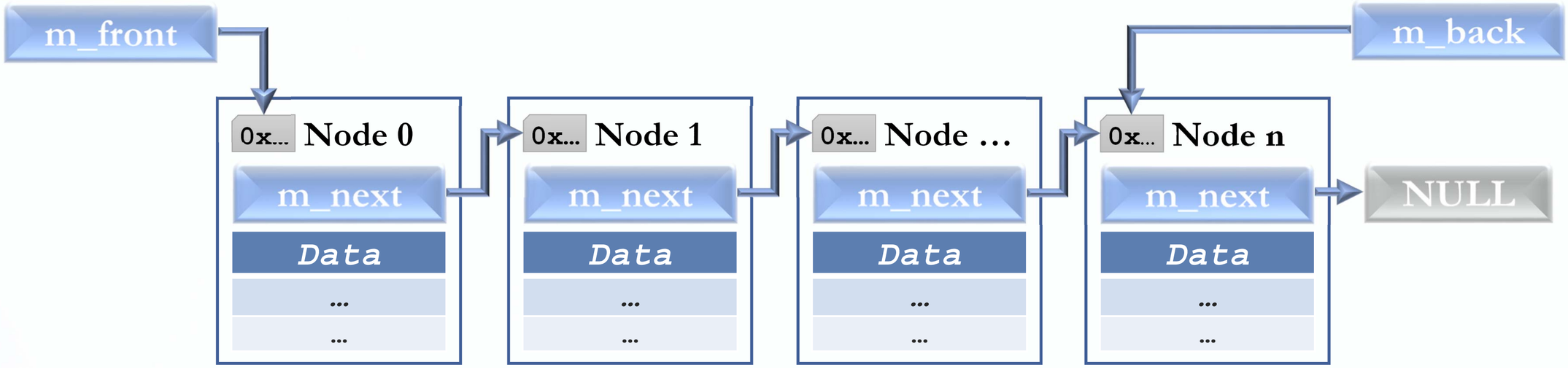
- Use a **`m_full`** flag to distinguish between the full and empty conditions.
- Declare **`m_maxsize+1`** locations for the array items, but use only **`m_maxsize`** of them for the Queue elements.

List-based Queue(s)

List-based Implementation(s)

A Queue can be implemented with a Linked List, as shown here.

- a) A Linear LL with 2 Pointers: *front* & *back*.

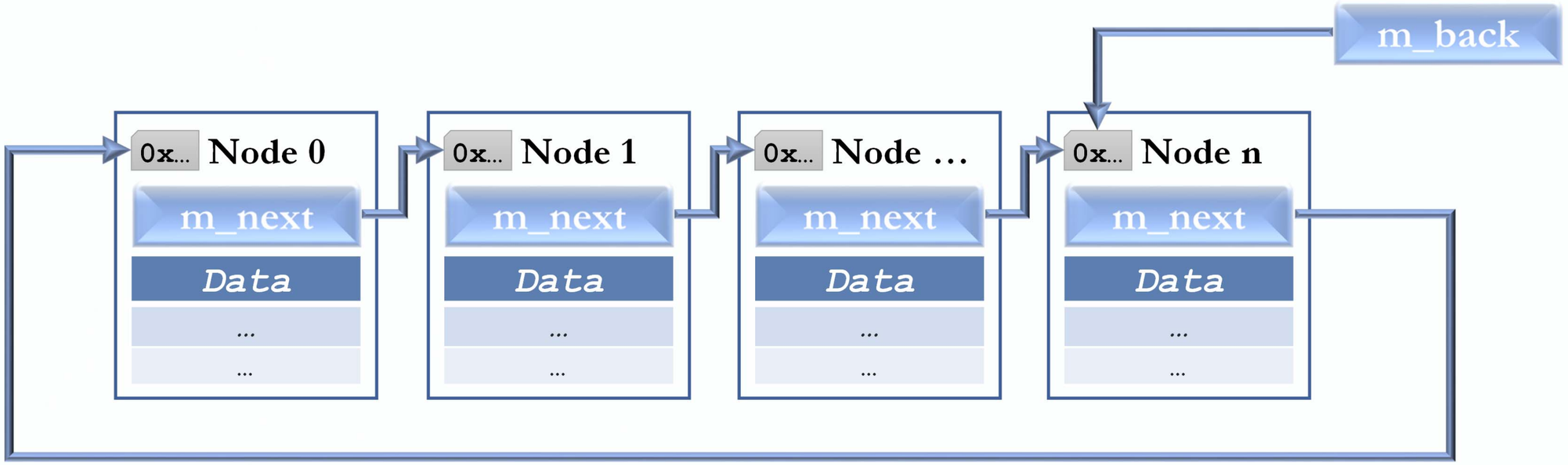


List-based Queue(s)

List-based Implementation(s)

A Queue can be implemented with a Linked List, as shown here.

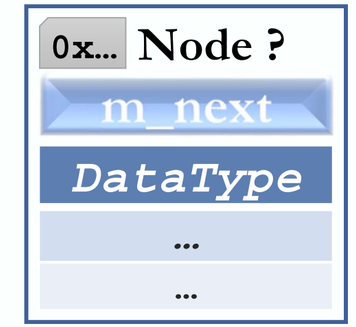
- b) A Circular Linear LL with 1 Pointer: *back*.

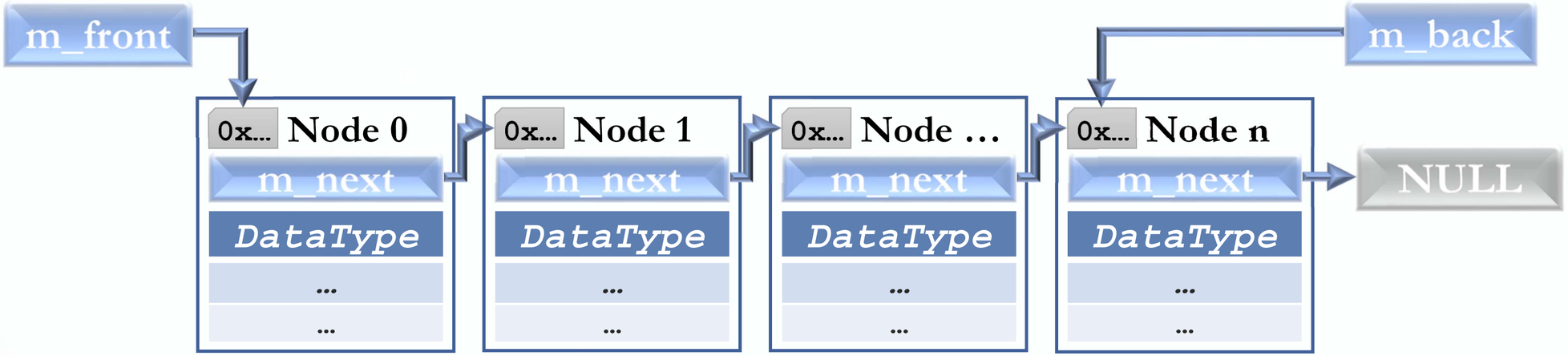


List-based Implementation(s)

A Queue **`push()`** (**enqueue**) operation.

Elements are exclusively pushed to the *back* of the Queue.



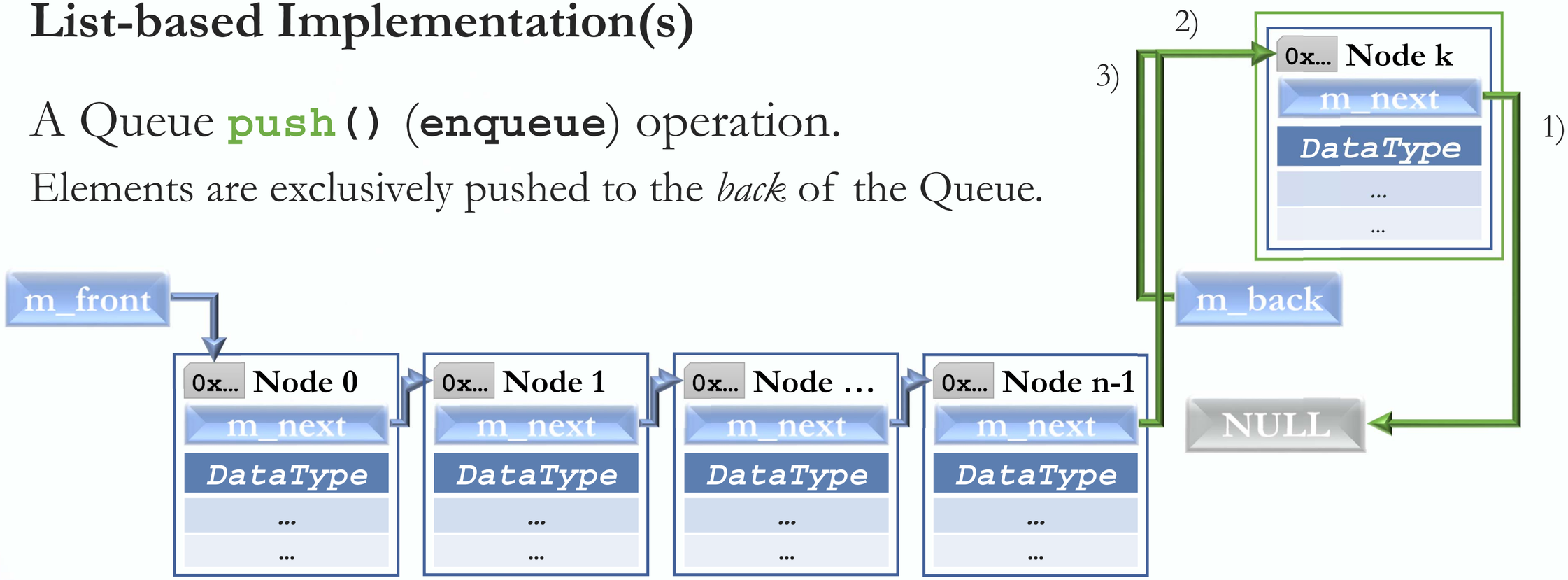


List-based Queue(s)

List-based Implementation(s)

A Queue **`push()`** (**enqueue**) operation.

Elements are exclusively pushed to the *back* of the Queue.



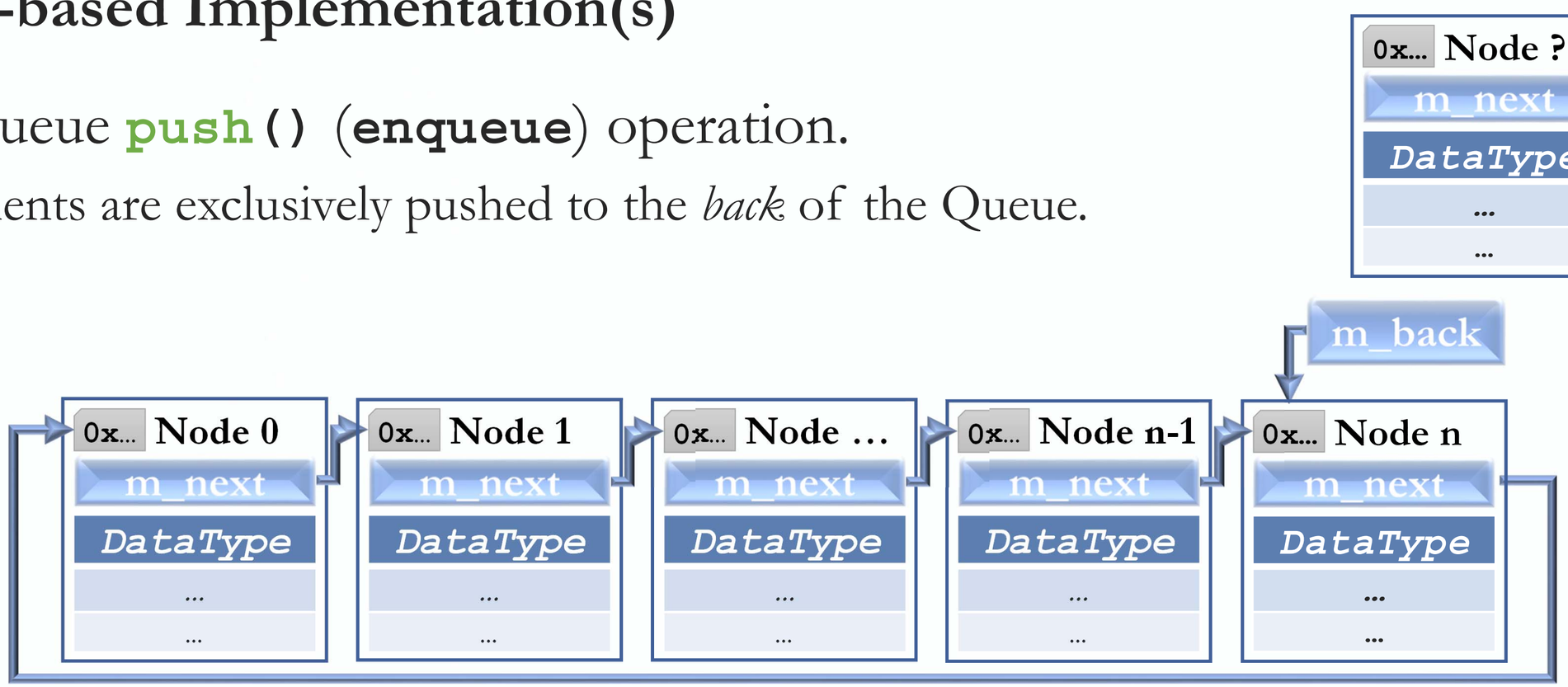

```
1) newNode_Pt->m_next = NULL;
2) m_back->m_next = newNode_Pt;
3) m_back = newNode_Pt;
```

List-based Queue(s)

List-based Implementation(s)

A Queue **`push()`** (**enqueue**) operation.

Elements are exclusively pushed to the *back* of the Queue.

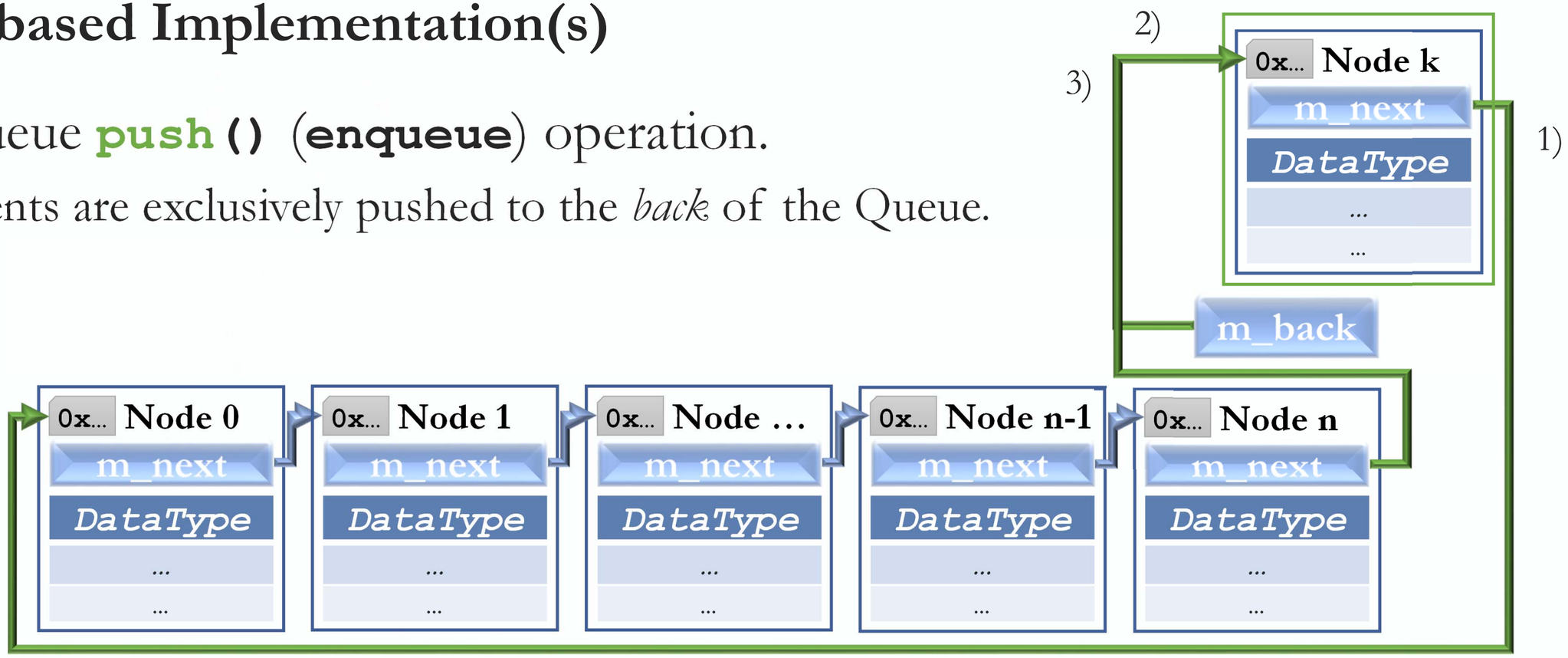


List-based Queue(s)

List-based Implementation(s)

A Queue **`push()`** (**enqueue**) operation.

Elements are exclusively pushed to the *back* of the Queue.

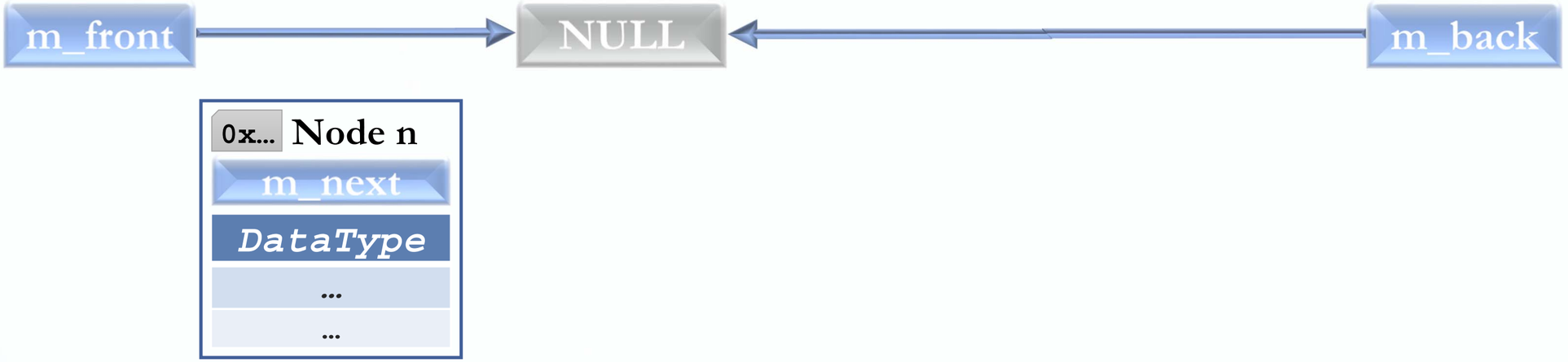


1) `newNode_Pt->m_next = m_back->m_next;`
2) `m_back->m_next = newNode_Pt;`
3) `m_back = newNode_Pt;`

List-based Implementation(s)

A Queue `push()` (**enqueue**) operation.

An originally empty Queue.

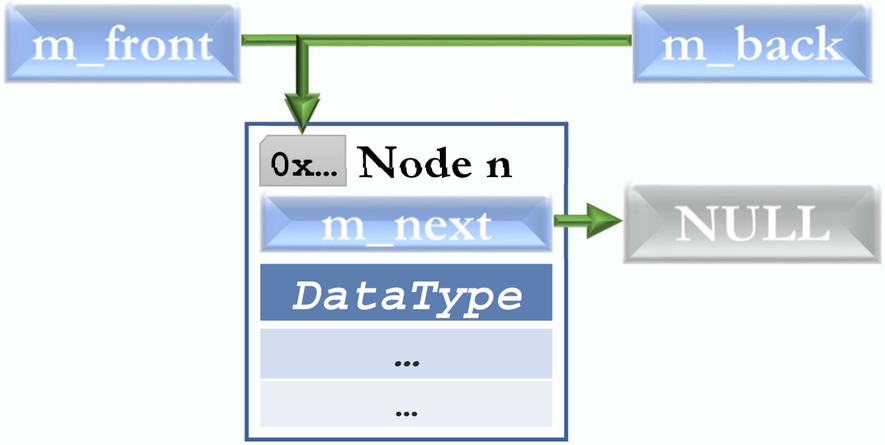


List-based Queue(s)

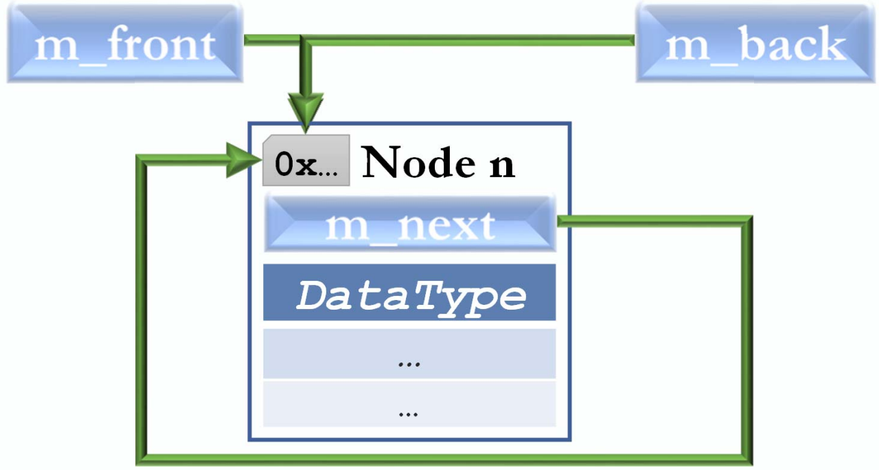
List-based Implementation(s)

A Queue **`push()`** (**enqueue**) operation.

An originally empty Queue.




```
1) newNode_Pt->m_next = NULL;
2) m_back = newNode_Pt;
3) m_front = newNode_Pt;
```



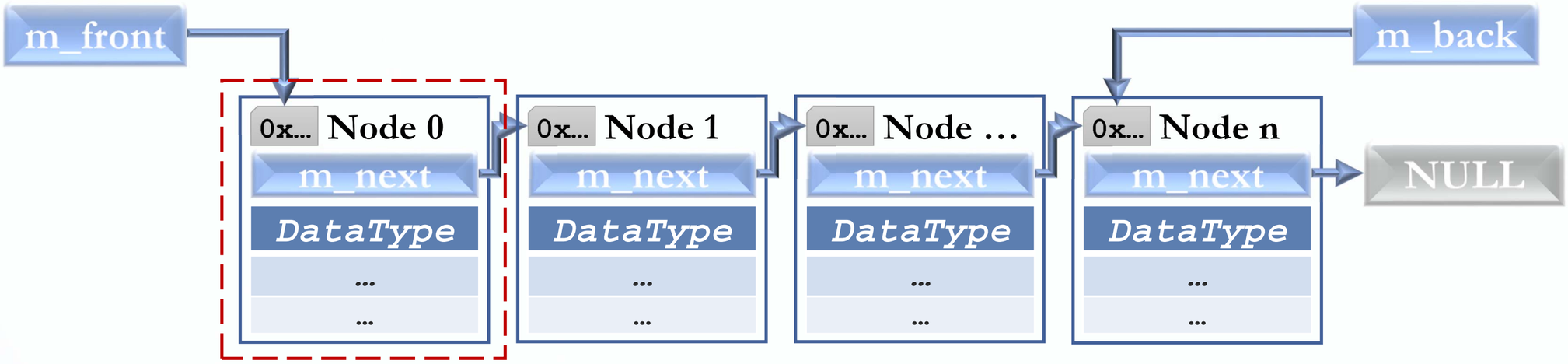

```
1) newNode_Pt->m_next = newNode_Pt;
2) m_back = newNode_Pt;
3) m_front = newNode_Pt;
```

List-based Queue(s)

List-based Implementation(s)

A Queue `pop()` (**`dequeue`**) operation.

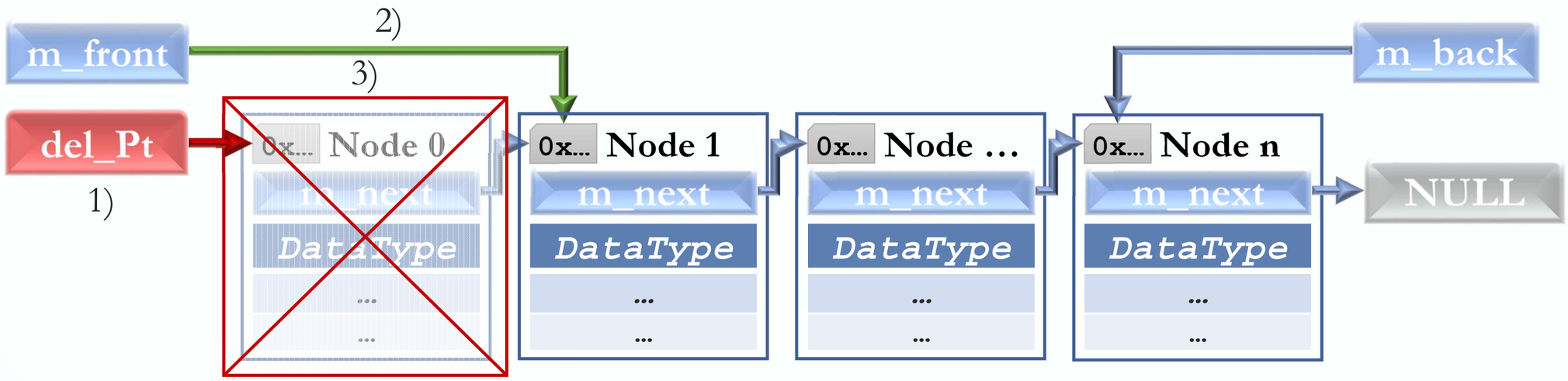
Elements are exclusively popped from the *front* of the Queue.



List-based Implementation(s)

A Queue `pop()` (**`dequeue`**) operation.

Elements are exclusively popped from the *front* of the Queue.

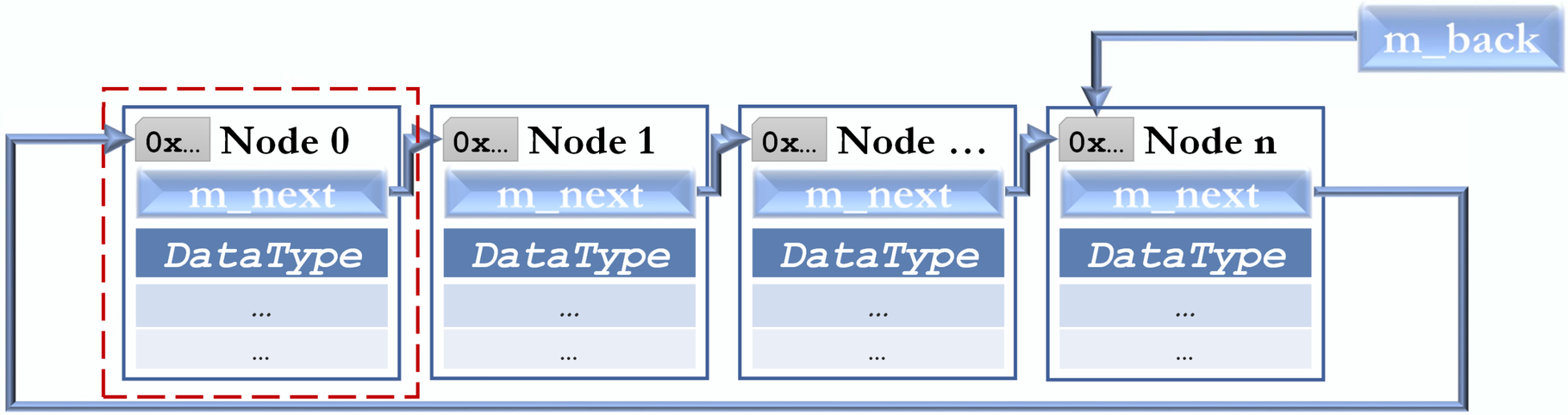



```
1) Node* del_Pt = m_front;
2) m_front = m_front->m_next;
3) delete del_Pt;
```

List-based Implementation(s)

A Queue `pop()` (**`dequeue`**) operation.

Elements are exclusively popped from the *front* of the Queue.

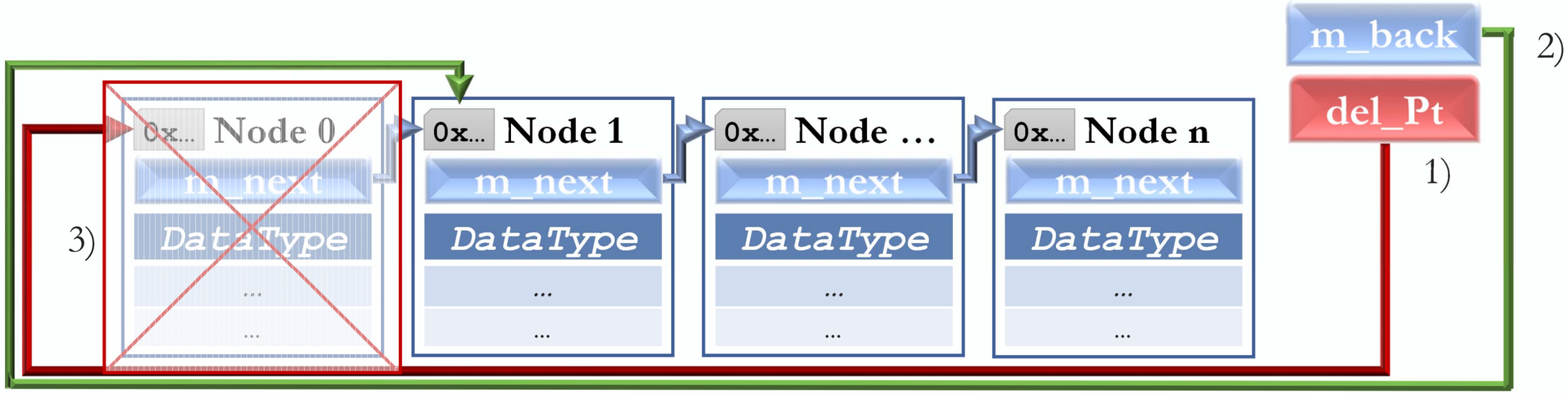


List-based Queue(s)

List-based Implementation(s)

A Queue `pop()` (**dequeue**) operation.

Elements are exclusively popped from the *front* of the Queue.




```
1) Node* del_Pt = m_back->m_next;
2) m_back->m_next = m_back->m_next->m_next;
3) delete del_Pt;
```

List-based Queue(s)

List-based Implementation(s)

```
typedef pod-or-class-or-struct-type DataType;
class Queue {
  public:
    Queue();
    Queue(int count, const DataType& val);
    Queue(const Queue& other);
    ~Queue();
    Queue& operator=(const Queue& other);
    bool empty() const;
    int size() const;
    void push(const DataType& value);
    void pop();
    void clear();
    DataType& front();
    DataType& back();
  private:
    QueueNode *m_front, *m_back;
    int       m_size;
};
```

```
class QueueNode {
  friend class Queue;
  public:
    QueueNode(){
      m_next = NULL;
    }
    QueueNode(const DataType& value,
              const QueueNode* next = NULL){
      m_Data = value;
      m_next = next;
    };
    const Data& data() const{
      return m_data;
    }
    Data& data() {
      return m_data;
    }
  private:
    DataType m_data;
    QueueNode* m_next;
}
```

Queue Applications

A "Palindrome"

- A string of characters that reads the same from left to right as its does from right to left.

  "Nipson anomemata me monan opsin"

  "ΝΙΨΟΝ ΑΝΟΜΗΜΑΤΑ ΜΗ ΜΟΝΑΝ ΟΨΙΝ"

Church of St. Mary of Blachernae

To recognize a palindrome, a Queue can be used in conjunction with a Stack.

- A Stack reverses the order of occurrences.
- A Queue preserves the order of occurrences.

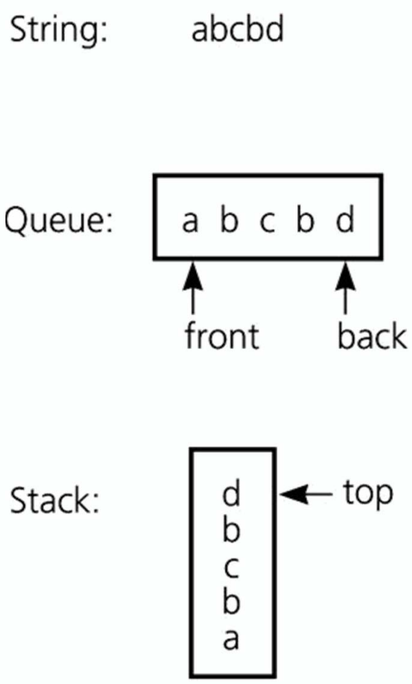
Queue Applications

Recognizing a "Palindrome" – example:

- A non-recursive recognition algorithm for palindromes.

As we traverse the character string from left to right:

- Insert each character into both a queue and a stack.
- Compare the characters at the front of the queue and the top of the stack.



Queue Applications

A "Simulation"

- A technique for modeling the behavior of both natural and human-made systems.

Goal

- Generate statistics that summarize the performance of an existing system.
- Predict the performance of a proposed system.

Queue Applications

A "Simulation" – example:

- A simulation of the behavior of a bank.

As customers arrive, they go to the back of the line:

- Use a Queue to represent the line of customers in the bank.
- The current customer, who is at the front of the line, is being served.
- This customer is followingly removed from the system.

Summary

Position-oriented ADTs

Position-oriented ADTs include:

- The List
- The Stack
- The Queue

Stacks and Queues

- Only the end positions/entries can be accessed.

Lists

- All positions/entries can be accessed.

Summary

Position-oriented ADTs

Operations of stacks and queues can be paired off as:

- **`createStack`** and **`createQueue`**.
- Stack **`empty`** and Queue **`empty`**.
- **`push`** and **`enqueue`**.
- **`pop`** and **`dequeue`**.
- Stack **`top`** and Queue **`front`**.

ADT List operations generalize Stack and Queue operations:

- **`size`** .
- **`insert`** .
- **`remove`** .
- **`retrieve`** .

CS-202

Time for Questions !