CS-202

Dynamic Data Structures (Pt.3)

C. Papachristos

Autonomous Robots Lab

University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	PASS Session		PASS Session

Your 8th Project will be announced today Thursday 4/11.

Smart Pointer(s) *extra-grade* Project X was this Wednesday 4/10.

- NO Project accepted past the 24-hrs delayed extension (@ 20% grade penalty).
- Send what you have in time!

Today's Topics

Dynamic Data Structures

"Container Adapters"

Queues(s)

- Array-based
- Node-based

Queue(s)

The Basics

A Dynamic Data Structure type, with its own semantics.

- "Adapts" the *interface* of a Container used for its backend.

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.

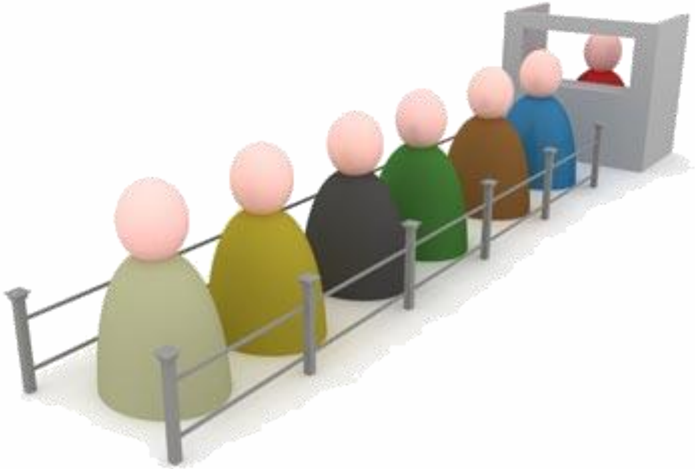
Queue(s)

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.



Queue(s)

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 – Concurrent programming (Note: Not the same as Multi-threading)
  For a printer – Serving a request to print a document.

Cross-field simulations & case-studies:

- ➢ Strategies to reduce the wait times involved in an application.
  (e.g. accounting for extra parameters such as wait time, multiple queues, etc.)
  https://coe.neu.edu/healthcare/pdfs/publications/intro_computer_simulation_healthcare_case_study.pdf

Queue(s)

Queue Operations

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

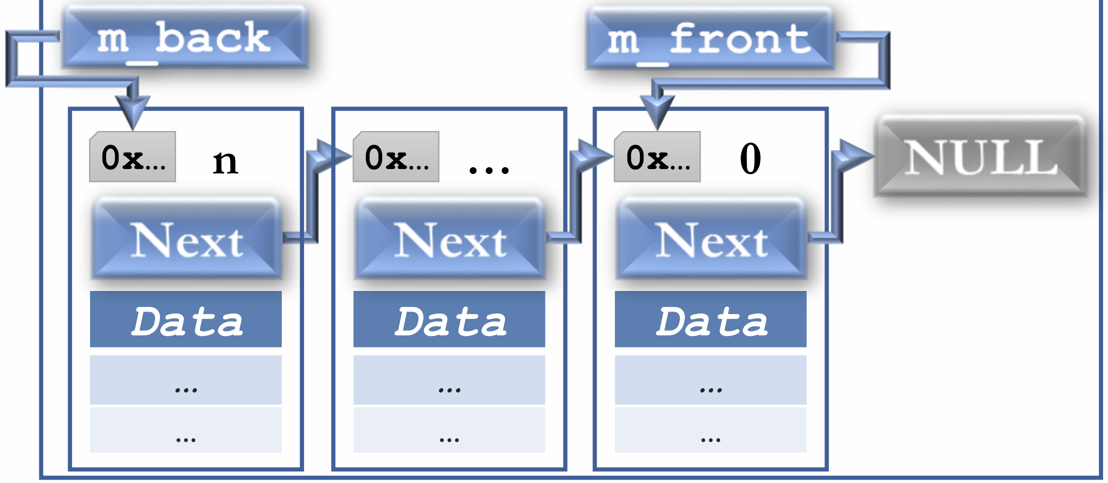
- Creation of an empty Queue.
- Destroying a Queue.
- Determining whether a Queue is empty.
- Adding (at the back) a new element to the Queue.
- Removal (from the front) of the item that was added earliest.
- Retrieval of the earliest added item (at the front).

Queue Operations

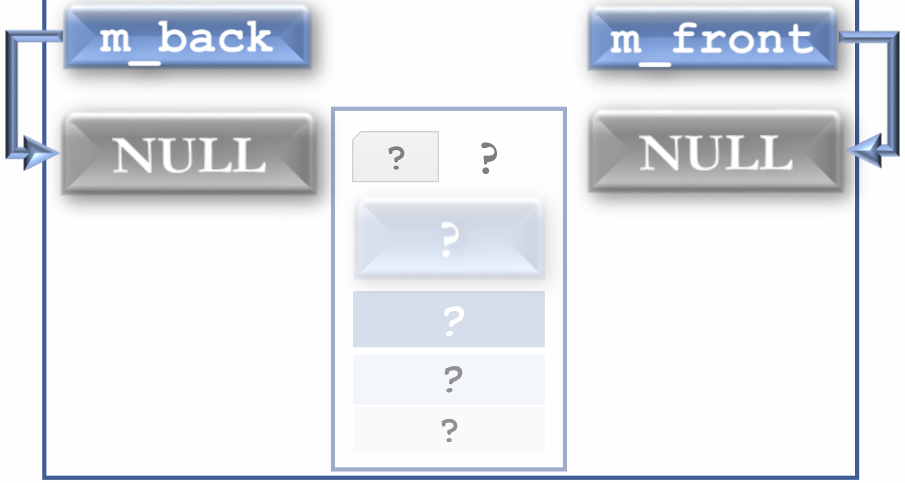
Queue empty()

```
bool empty() const;
/** Determines whether the queue is empty.
 * @return True if the queue is empty, otherwise false.
 */
```

Non-empty Queue: false



Empty Queue: true



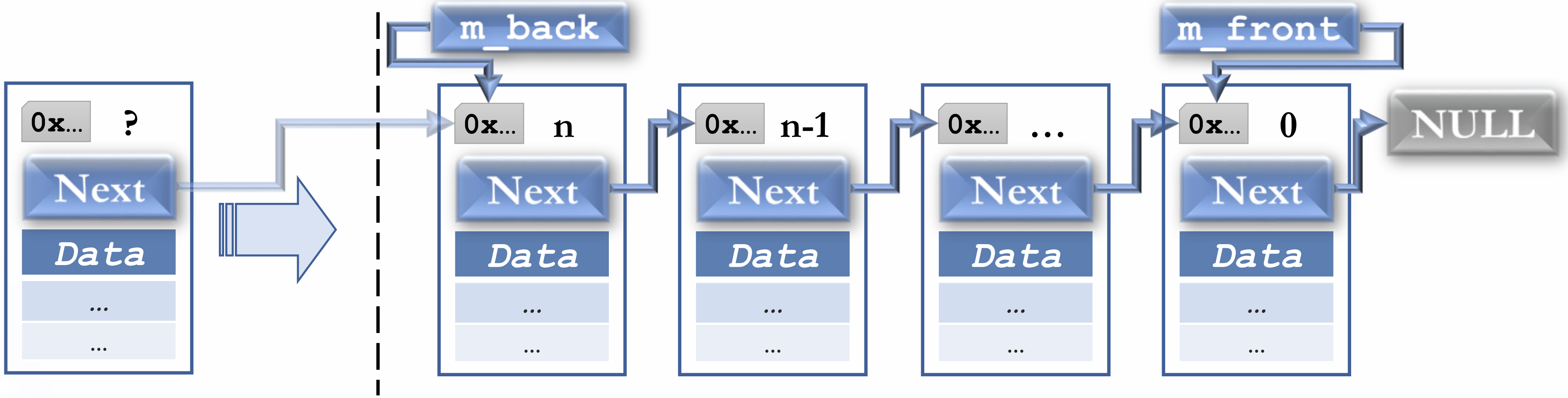
Queue Operations

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.

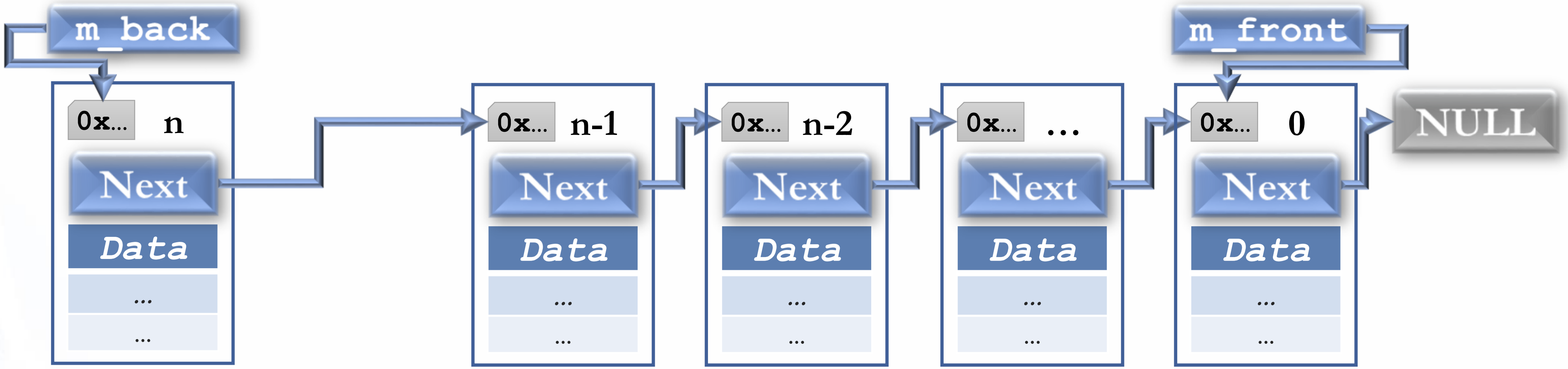
Called an "**`enqueue`**" operation (also **`push`**, **`addElement`**, etc.)



Queue Operations

Queue push()-ing

```
void push(const DataType & value);
/** Inserts an element at the back of a queue.
 * @param value The value of the element to be inserted.
 * @post If the insertion is successful, a new DataType element of value is
at the back of the queue.
 */
```



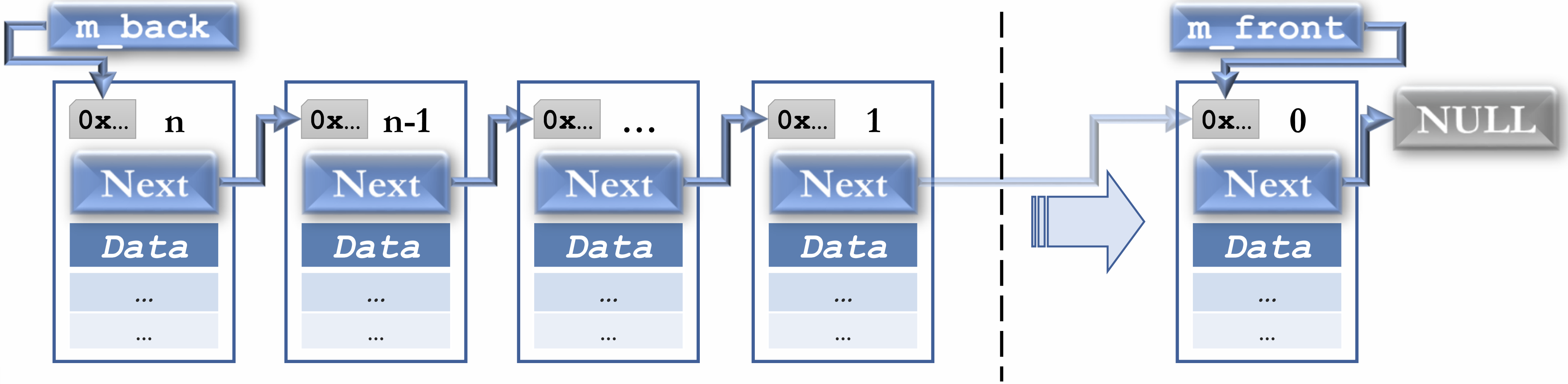
Queue Operations

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.

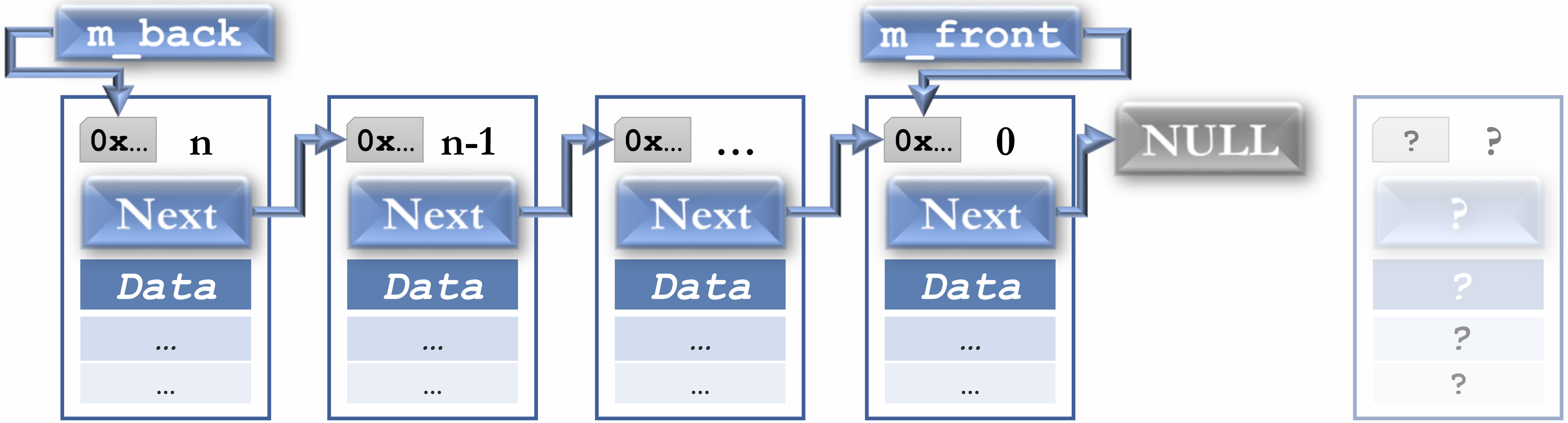
Called an "**dequeue**" operation (also **pop**, **removeElement**, etc.)



Queue Operations

Queue pop()-ing

```
void pop();
/** Dequeues the front of a queue.
 * @post If the queue is not empty, the element that was added to the queue
earliest is deleted.
*/
```



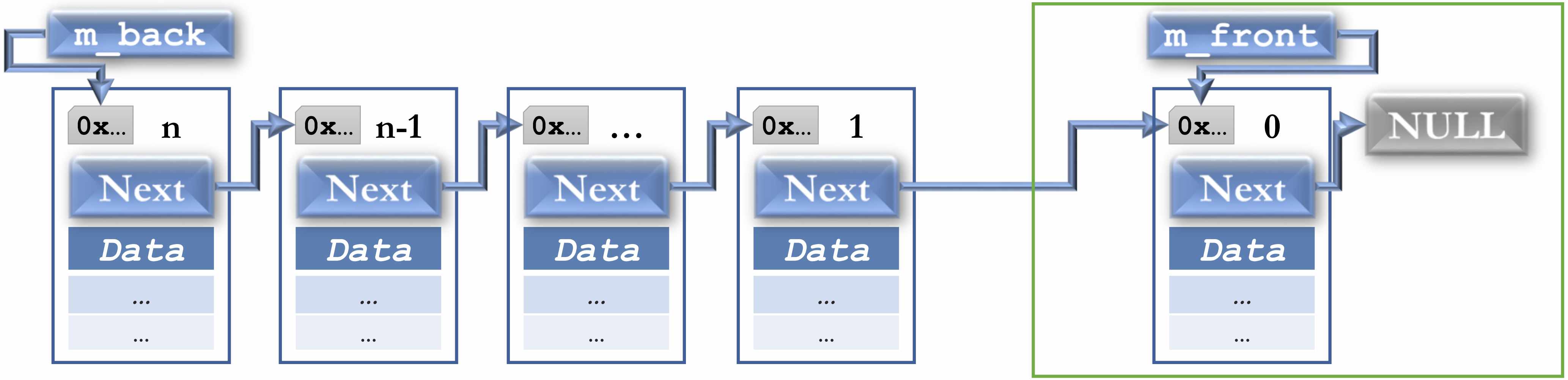
Queue Operations

Queue `front()`

When the front element needs to be accessed:

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

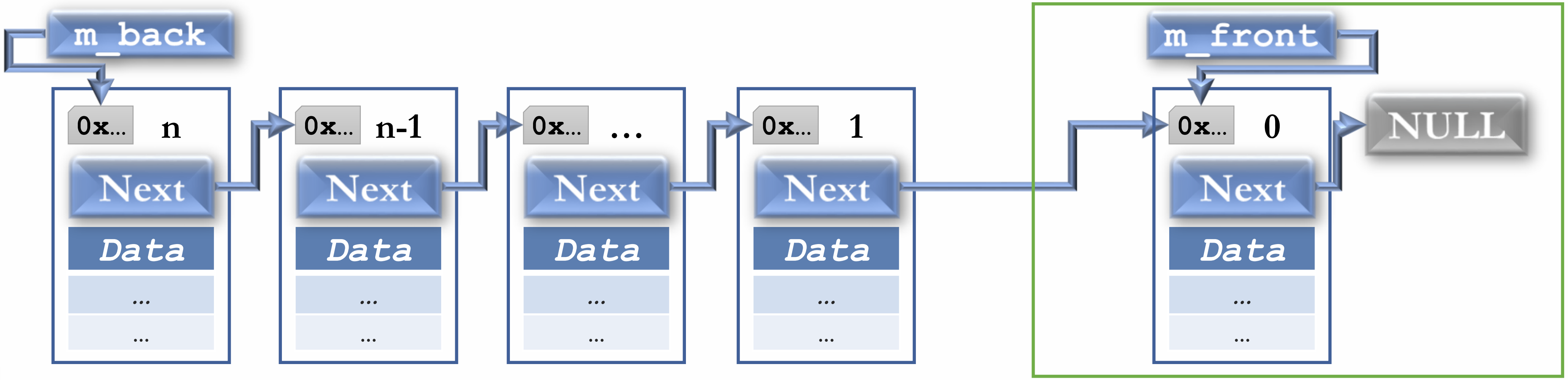
Called an "**`getFront`**" operation (also **`frontElement`**, etc.)



Queue Operations

Queue front()

```
DataType & front();
const DataType & front() const;
/** Retrieves the element at the front of a queue
 * @pre The queue is not empty.
 * @return If the queue is not empty, the return value is a (const) reference
to the earliest added element. Otherwise result is undefined.
*/
```



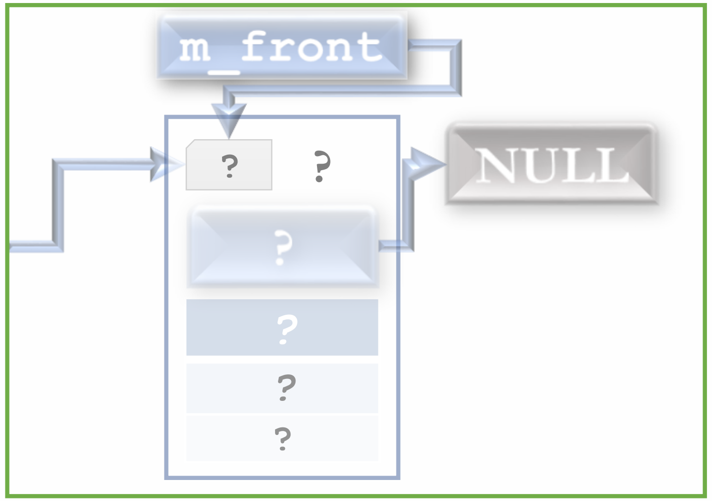
Queue Operations

Queue front()

```
DataType & front();
const DataType & front() const;
/** Retrieves the element at the front of a queue
 * @pre The queue is not empty.
 * @return If the queue is not empty, the return value is a (const) reference
to the earliest added element. Otherwise result in undefined.
*/
```

- 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.



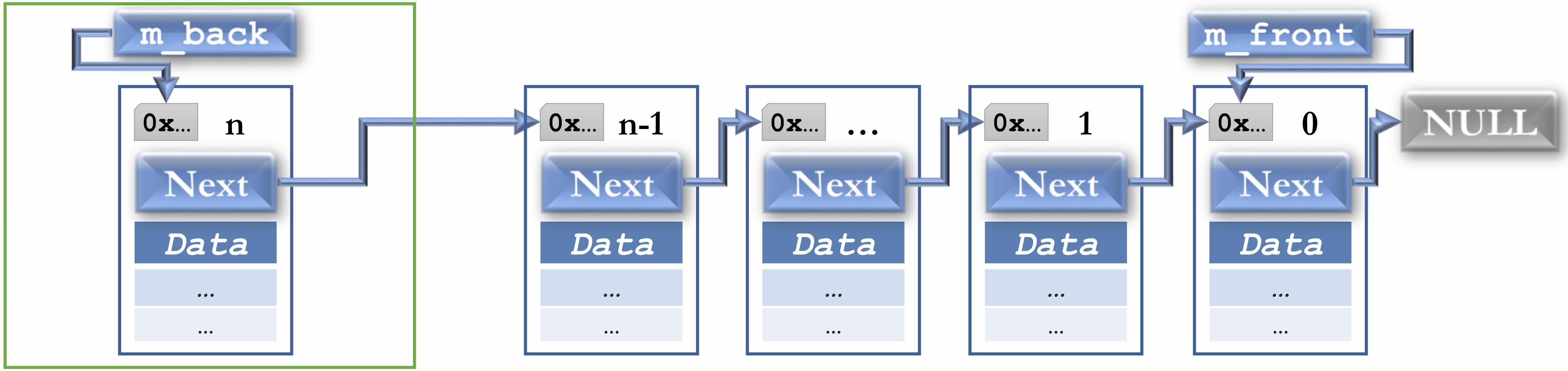
Queue Operations

Queue `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.

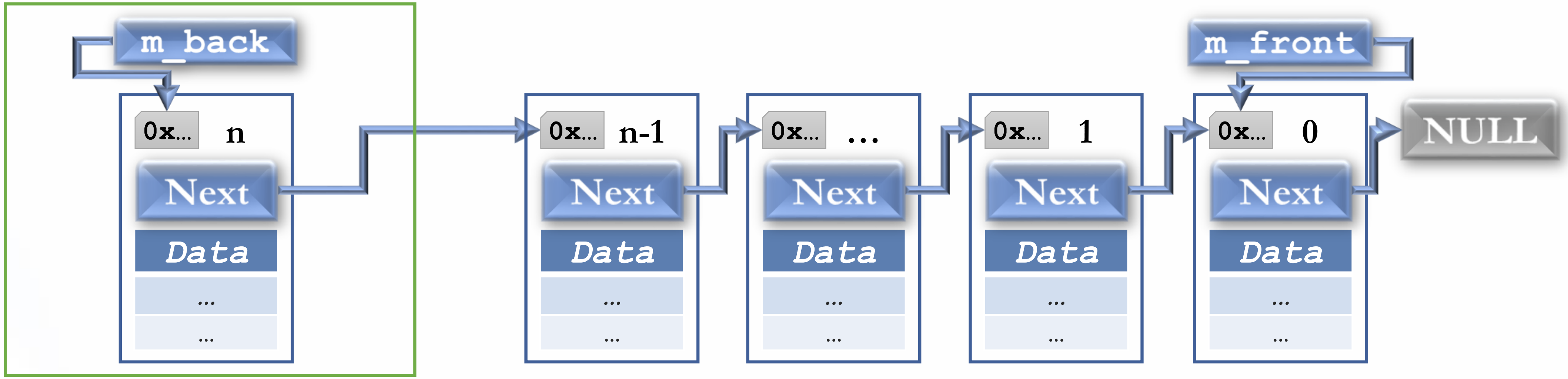
Called an "**`getBack`**" operation (also **`back`**, **`lastElement`**, etc.)



Queue Operations

Queue back() (outside of specifications)

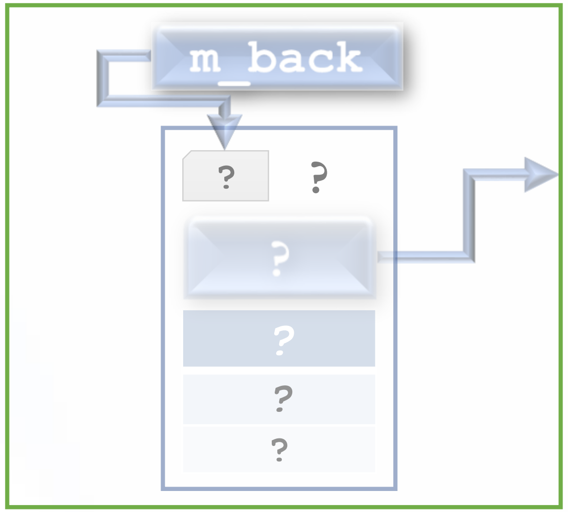
```
DataType & back();
const DataType & back() const;
/** Retrieves the element at the end of a queue
 * @pre The queue is not empty.
 * @return If the queue is not empty, the return value is a (const) reference
to the earliest added element. Otherwise result is undefined.
 */
```



Queue Operations

Queue back() (outside of specifications)

```
DataType & back();
const DataType & back() const;
/** Retrieves the element at the end of a queue
 * @pre The queue is not empty.
 * @return If the queue is not empty, the return value is a (const) reference
to the last added element. Otherwise result is undefined.
 */
```



- 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 “*Sequential Container*” ADT functionalities.

- A Queue is a “*Container Adapter*” and can have :
  - An Array-based backend / implementation.
  - A List (Node)-based backend / implementation.

    A Linked-List with two-ended access :
    - A pointer to the front element.
    - A pointer to the back element.

Array-based Queue(s)

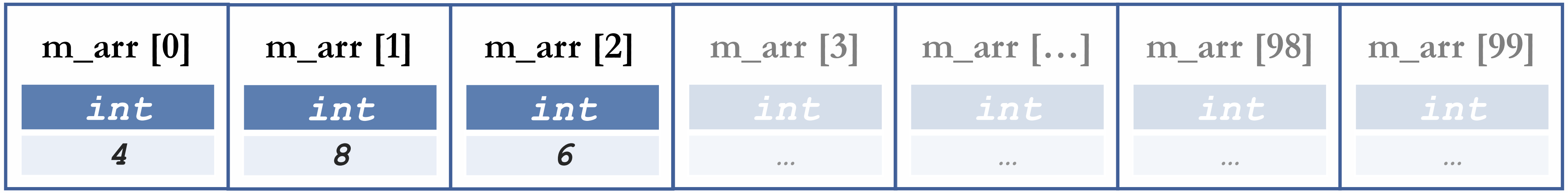
Array-based Implementation(s)

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)

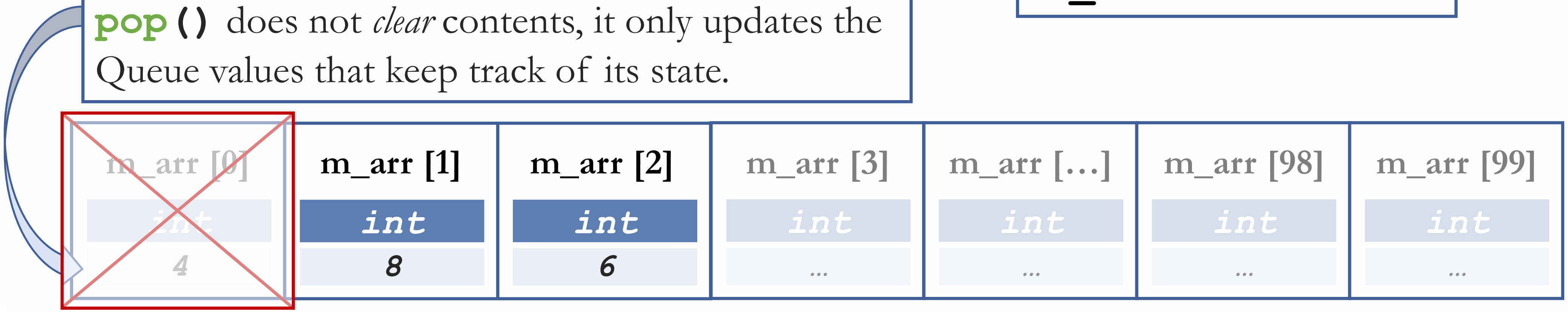
A Queue **pop()** (**dequeue**) operation – *Naïve* approach.

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.



~~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)

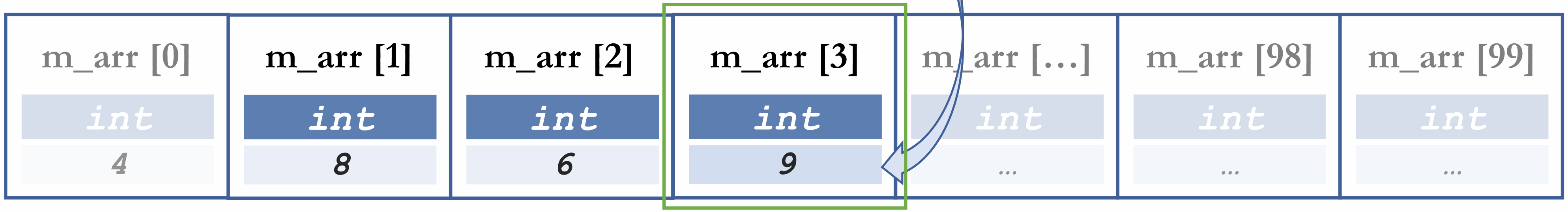
A Queue **push()** (**enqueue**) operation – *Naïve* approach.

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.



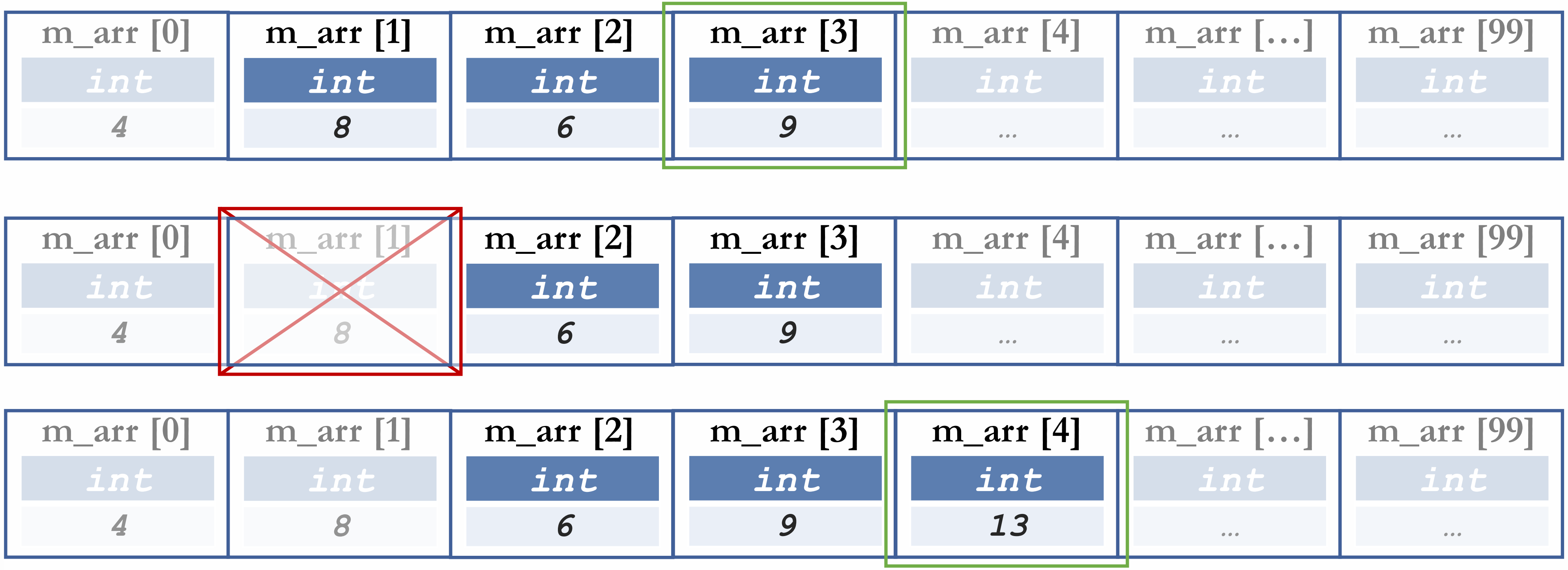
m_arr [0]	m_arr [1]	m_arr [2]	m_arr [3]	m_arr […]	m_arr [98]	m_arr [99]
int	*int*	*int*	*int*	*int*	*int*	*int*
4	*8*	*6*	*9*	…	…	…

Array-based Queue(s)

Array-based Implementation(s)

Queue *Naïve* approach issues.

For a sequence of operations: ADADADADADADADA… (A:Add, D: Delete)



m_arr [0]	m_arr [1]	m_arr [2]	m_arr [3]	m_arr [4]	m_arr […]	m_arr [99]
int	int	int	int	int	int	int
4	8	6	9	…	…	…

m_arr [0]	m_arr [1]	m_arr [2]	m_arr [3]	m_arr [4]	m_arr […]	m_arr [99]
int	int	int	int	int	int	int
4	8	6	9	…	…	…

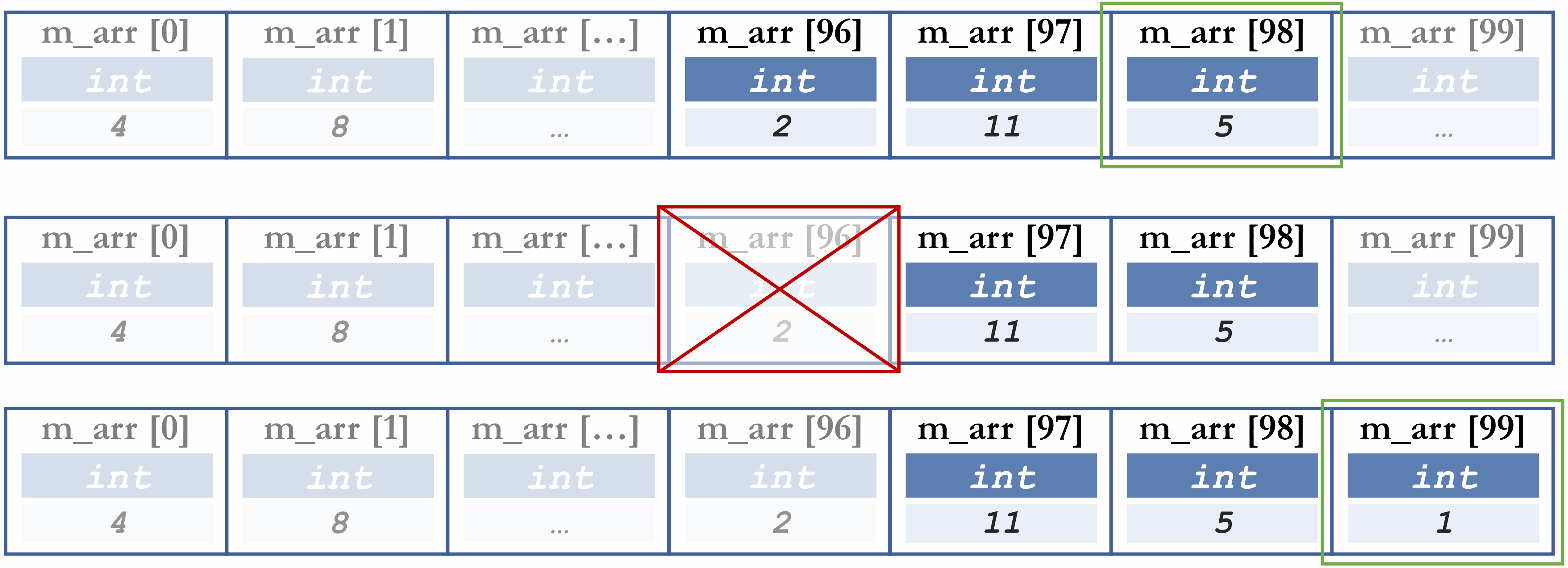
m_arr [0]	m_arr [1]	m_arr [2]	m_arr [3]	m_arr [4]	m_arr […]	m_arr [99]
int	int	int	int	int	int	int
4	8	6	9	13	…	…

Array-based Queue(s)

Array-based Implementation(s)

Queue *Naïve* approach issues.

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



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	…

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	…

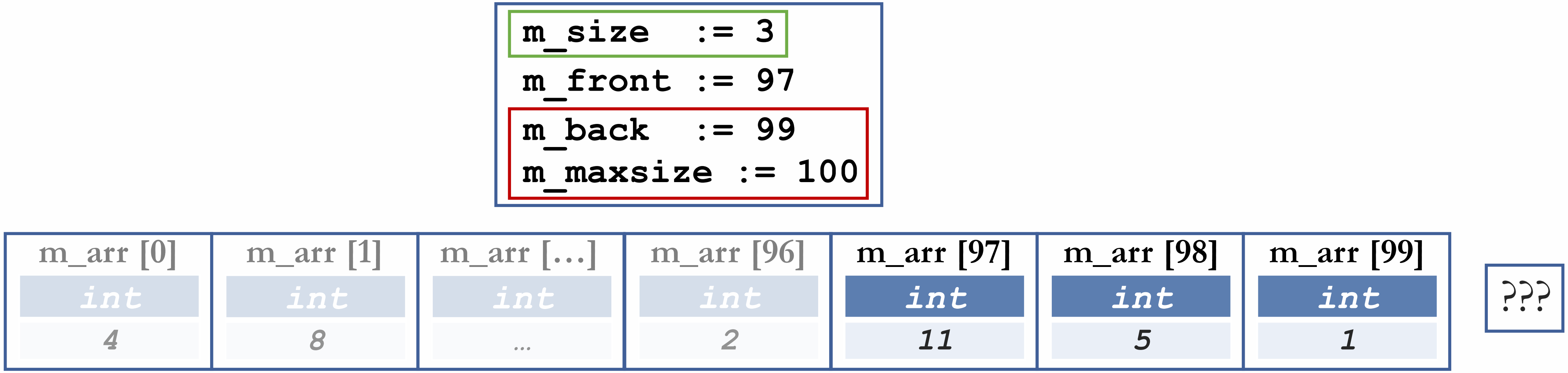
m_arr [0]	m_arr [1]	m_arr […]	m_arr [96]	m_arr [97]	m_arr [98]	m_arr [99]
int	int	int	int	int	int	int
4	8	…	2	11	5	1

Array-based Queue(s)

Array-based Implementation(s)

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*

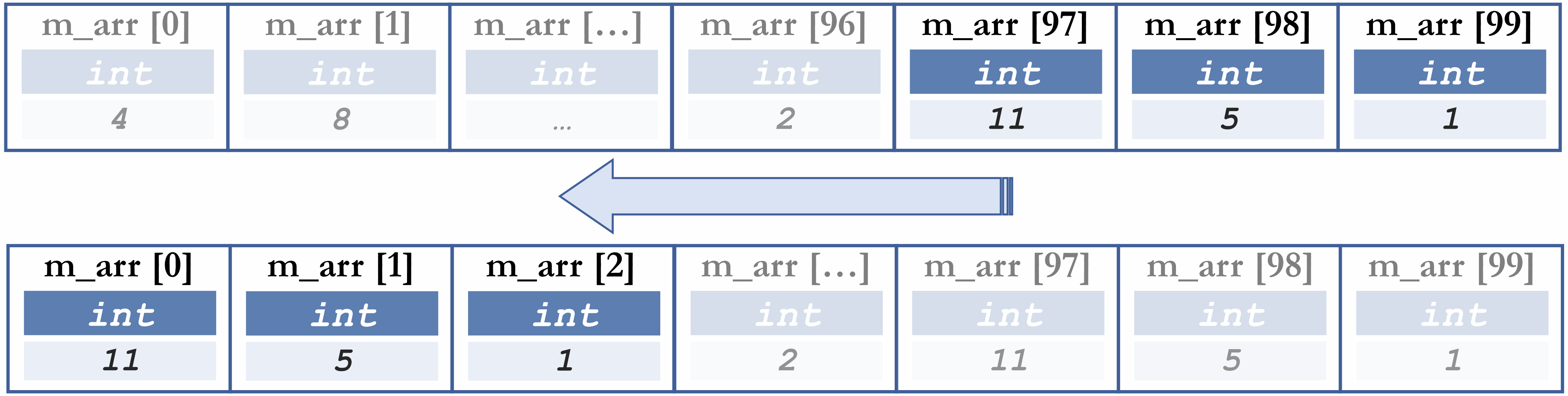
???

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.



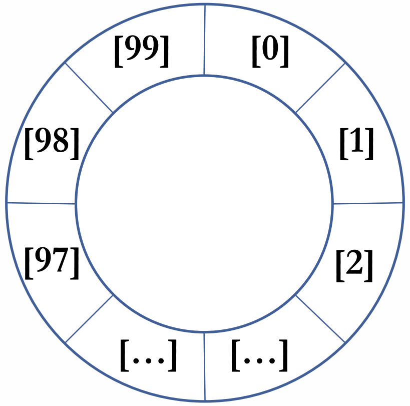
m_arr [0]	m_arr [1]	m_arr […]	m_arr [96]	m_arr [97]	m_arr [98]	m_arr [99]
int	int	int	int	int	int	int
4	8	…	2	11	5	1

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

Array-based Queue(s)

Array-based Implementation(s)

An “elegant” solution – The circular buffer 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 := 100   m_back  := 99
```

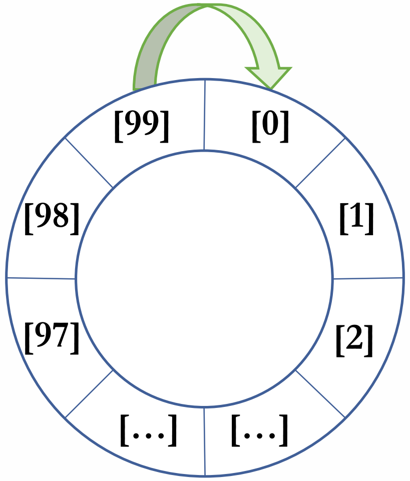
Advance m_back to next circular array position !

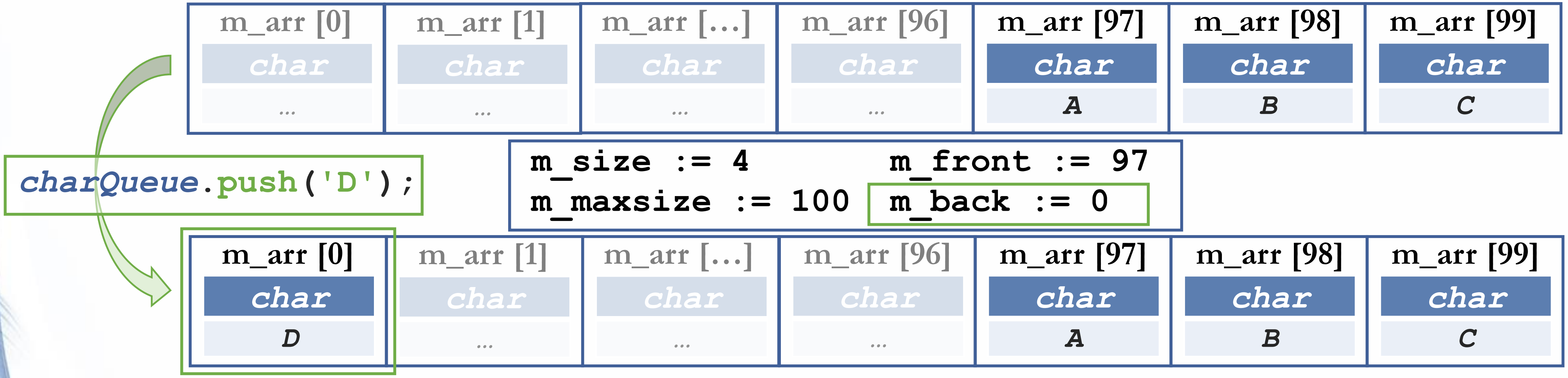
```
m_back = (m_back + 1) % m_maxsize;
```

Array-based Queue(s)

Array-based Implementation(s)

An “elegant” solution – The circular buffer 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    := 4       m_front := 97
m_maxsize := 100     m_back  := 0
```

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
D	…	…	…	A	B	C

Array-based Queue(s)

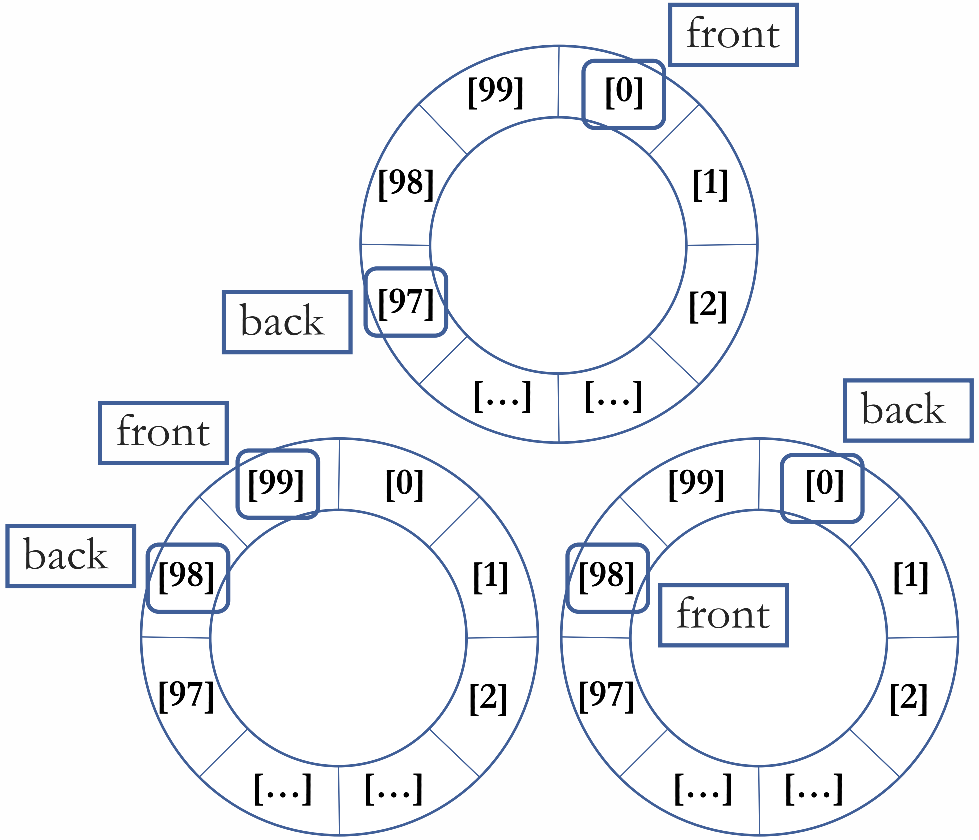
Array-based Implementation(s)

The circular buffer:

- Eliminates issue of rightward drift.

But:

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

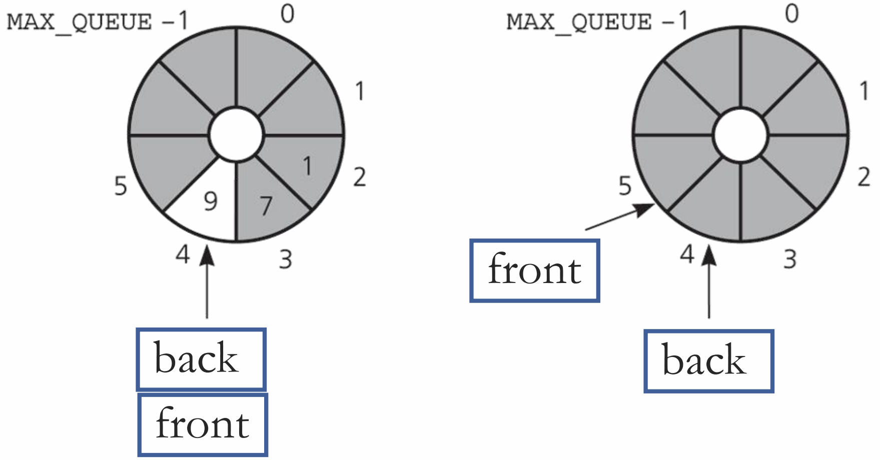


Array-based Queue(s)

Array-based Implementation(s)

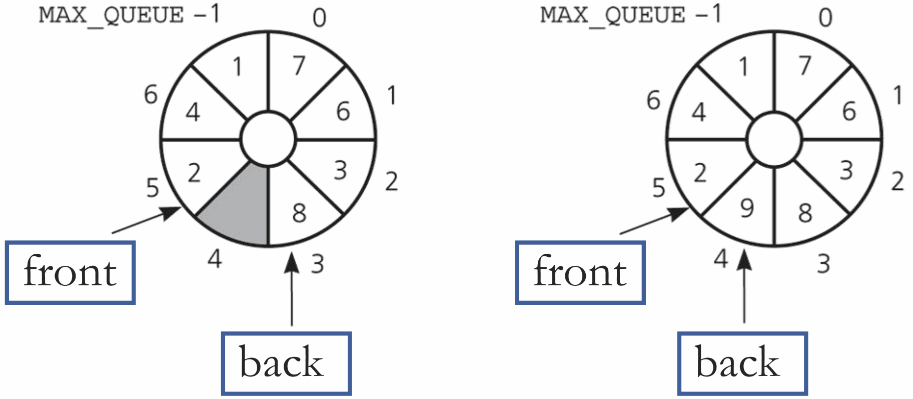
a) *front* passes *back*
when the queue becomes empty.

- ➢ 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.

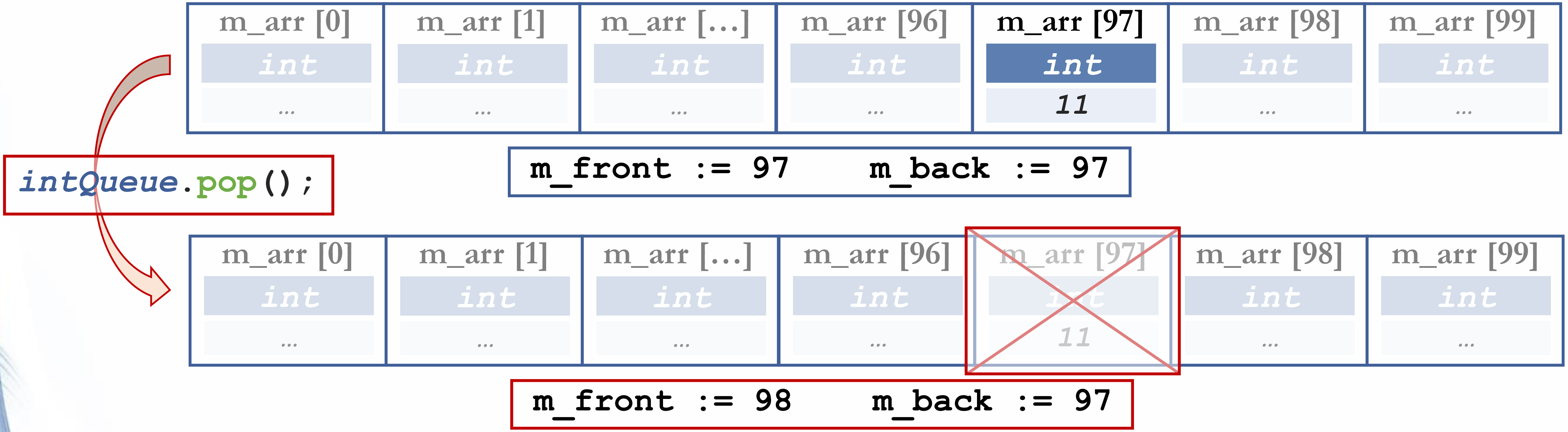


Array-based Queue(s)

Array-based Implementation(s)

Circular array issues (continued):

- ➢ Cases with identical *front* & *back* index values.
  An empty Queue, after a **dequeue** operation:



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
…	…	…	…	11	…	…

`m_front := 97     m_back := 97`

```
intQueue.pop();
```

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
…	…	…	…	11	…	…

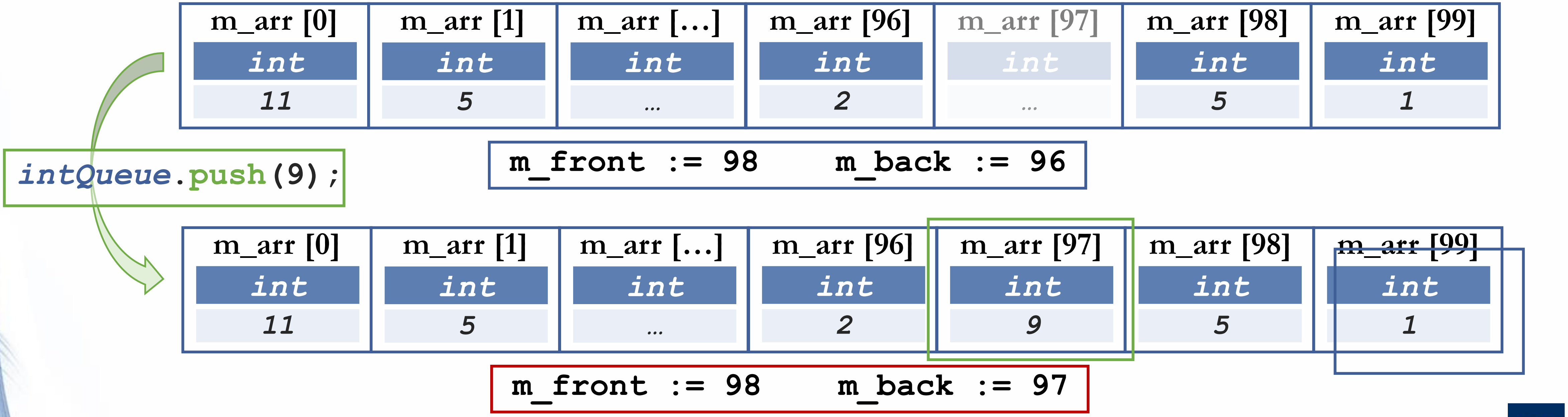
`m_front := 98     m_back := 97`

Array-based Queue(s)

Array-based Implementation(s)

Circular array issues (continued):

- Cases with identical *front* & *back* index values.
  A full Queue, after an **enqueue** operation:



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
11	5	…	2	…	5	1

```
m_front := 98     m_back := 96
```

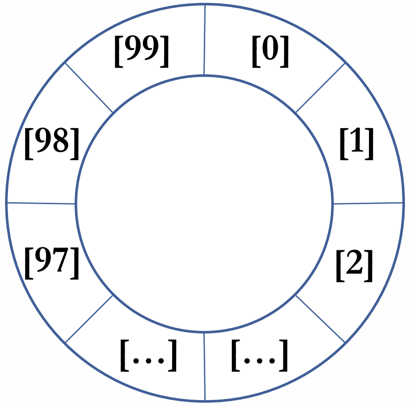
```
intQueue.push(9);
```

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
11	5	…	2	9	5	1

```
m_front := 98     m_back := 97
```

Array-based Queue(s)

Array-based Implementation(s)



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

- ➢ Keep a count of the queue elements (**`m_size`**).
- ➢ Increment when new element **`push`**’ed.
- ➢ Decrement when element **`pop`**’ped.

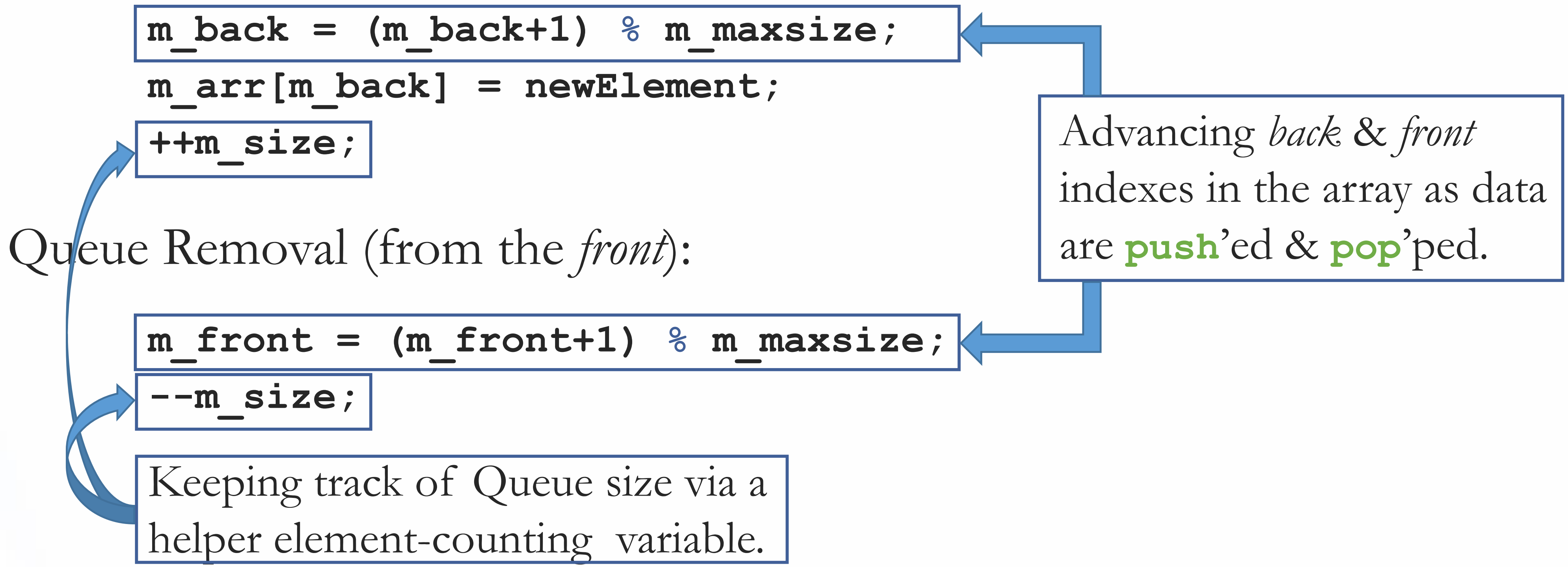
Queue Initialization:

- ➢ 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;
```

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

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

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;
    size_t size() const;
    void push(const DataType & value);
    void pop();
    void clear();
    DataType & front();
    DataType & back();
  private:
    DataType * m_arr;
    size_t     m_front, m_back;
    size_t     m_size, m_maxsize;
};
```

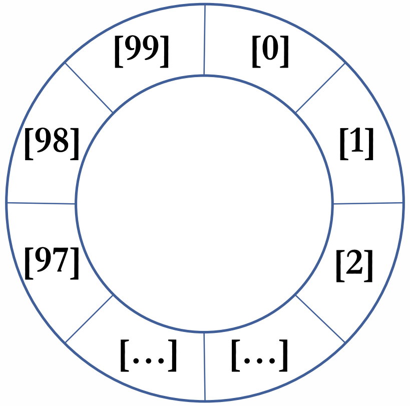
Array-based Queue(s)

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:

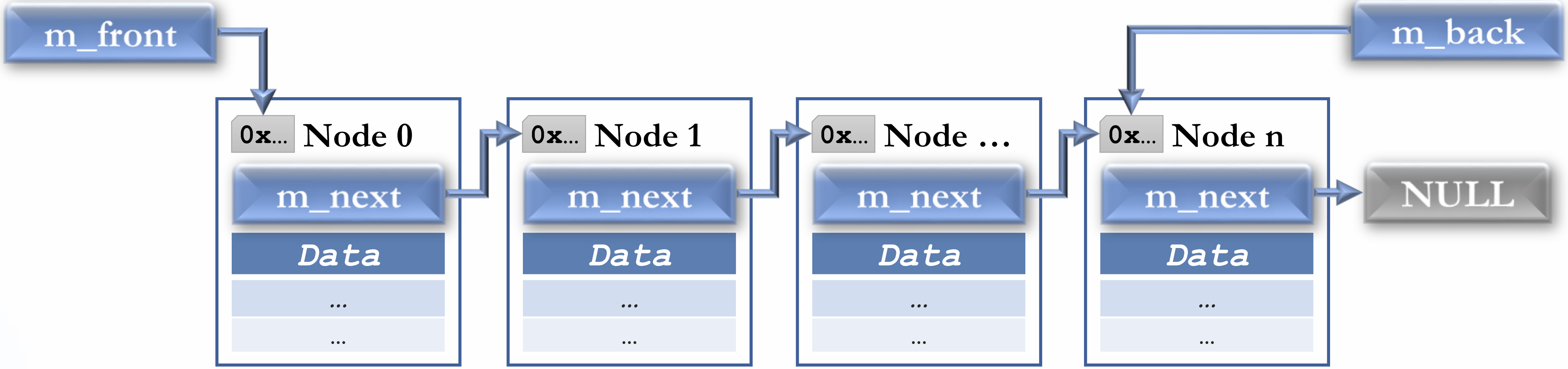
- ➢ Use a **`m_full`** flag to distinguish between the full and empty conditions.

List-based Queue(s)

List-based Implementation(s)

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

- a) A Linked–List with 2 Pointers: *front* & *back*.

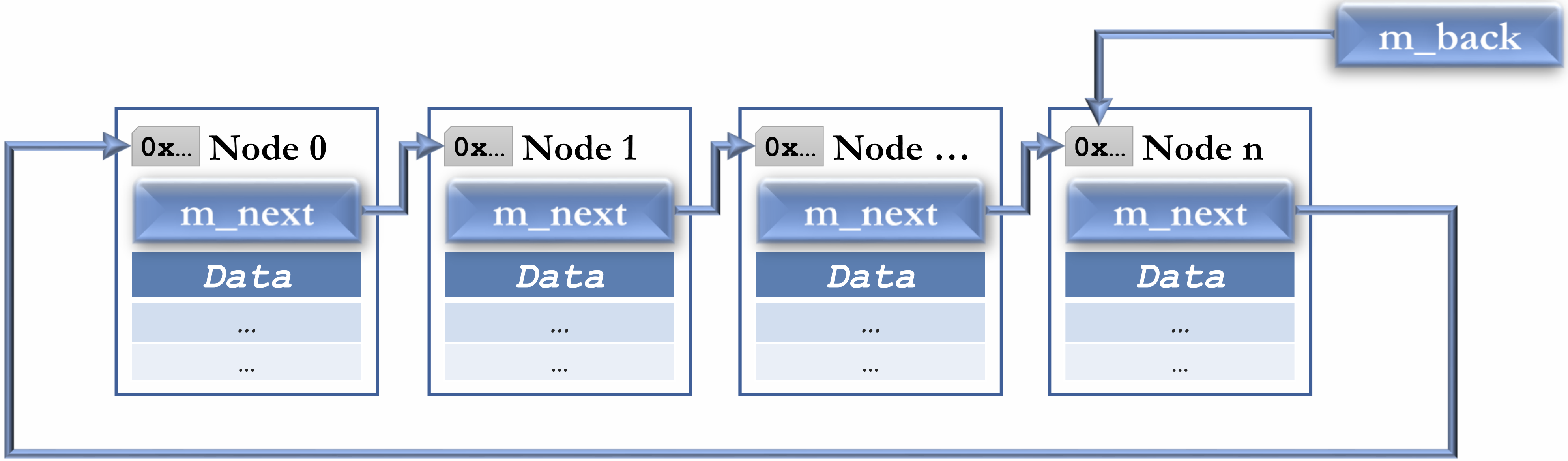


List-based Queue(s)

List-based Implementation(s) – *Unusual Case*

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

- b) A Circular Linked–List with 1 Pointer: *back*.

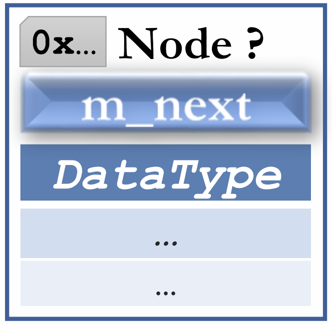


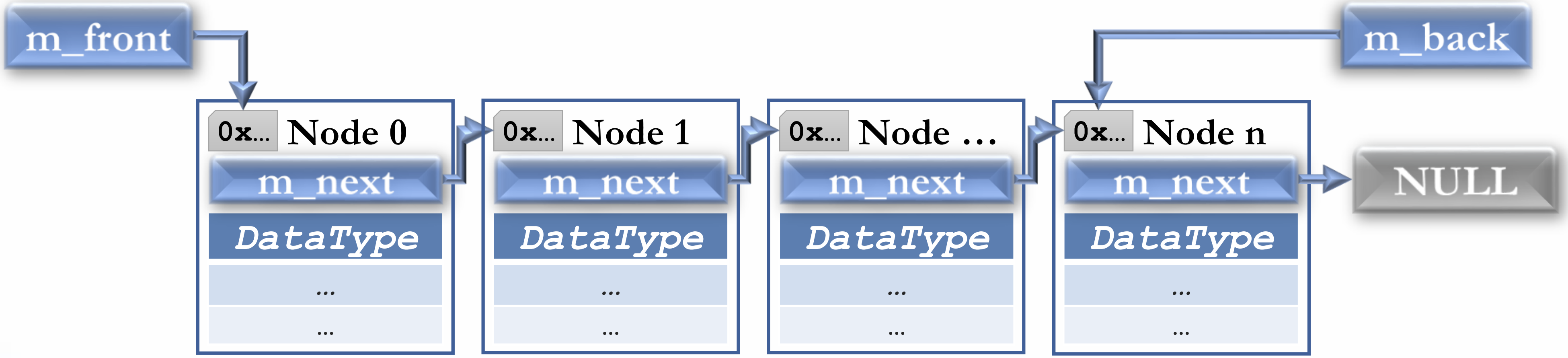
List-based Queue(s)

List-based Implementation(s)

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

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



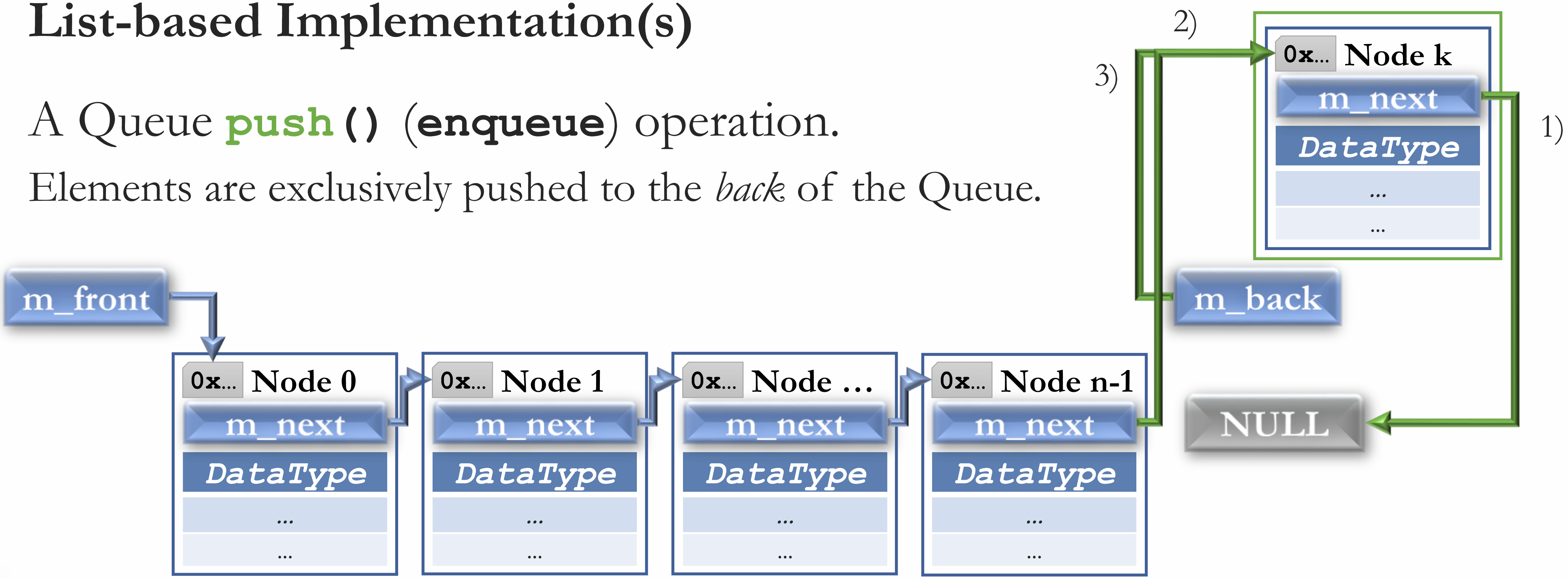


List-based Queue(s)

List-based Implementation(s)

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

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



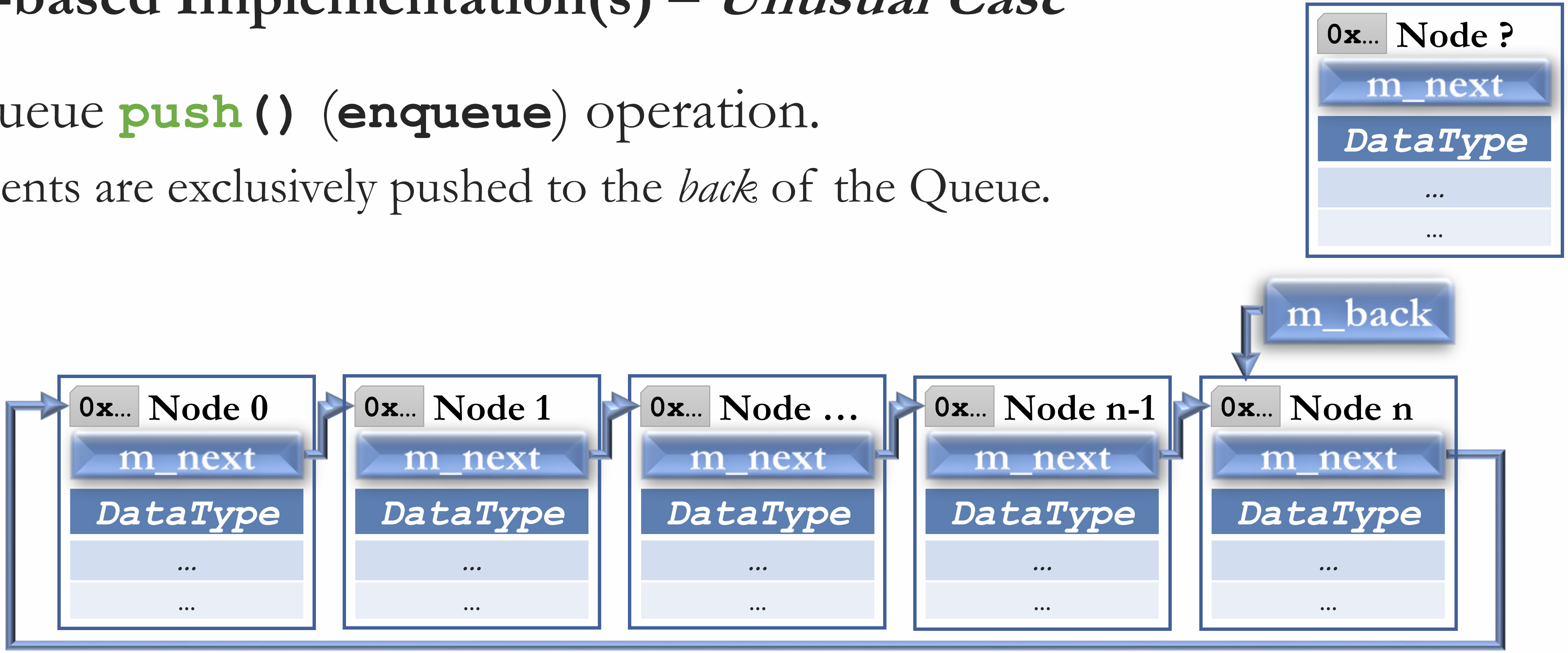
1) `newNode_Pt->m_next = NULL;`
2) `m_back->m_next = newNode_Pt;`
3) `m_back = newNode_Pt;`

List-based Queue(s)

List-based Implementation(s) – *Unusual Case*

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

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

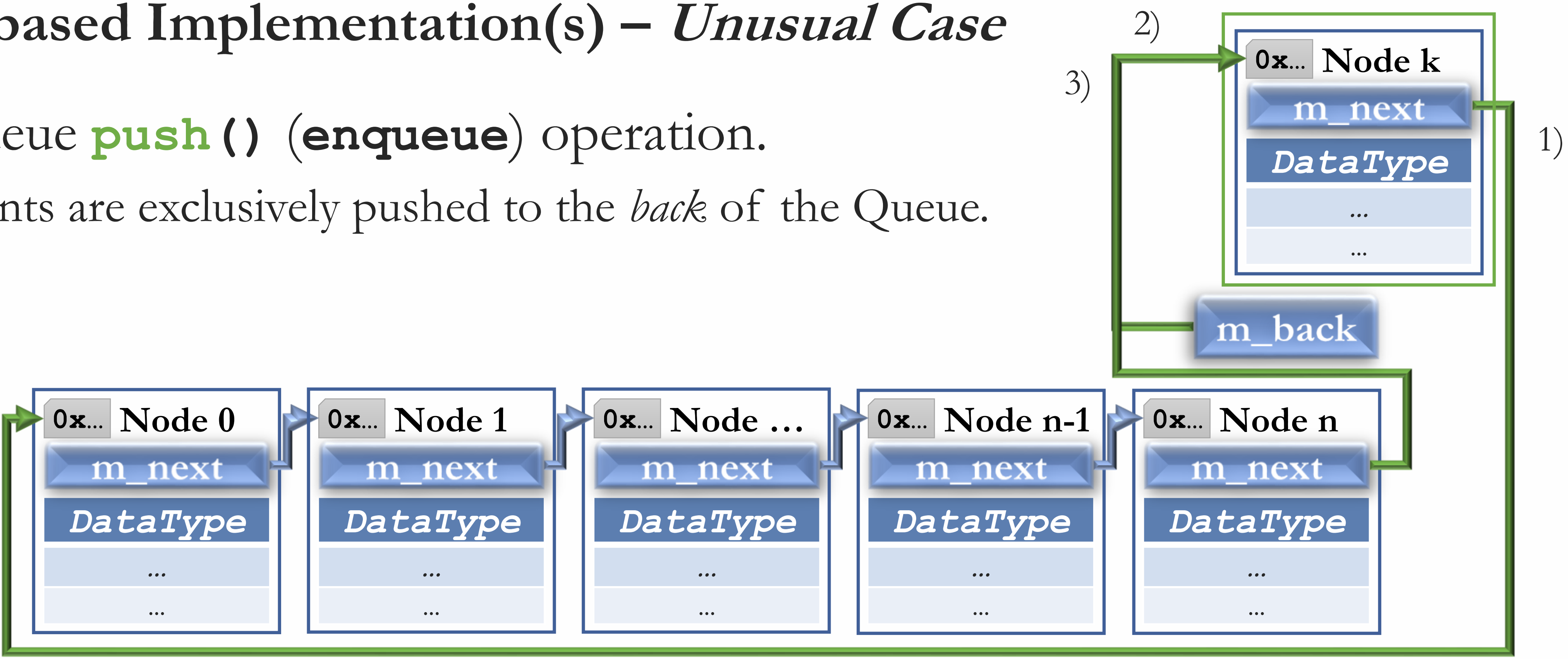


List-based Queue(s)

List-based Implementation(s) – *Unusual Case*

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;`

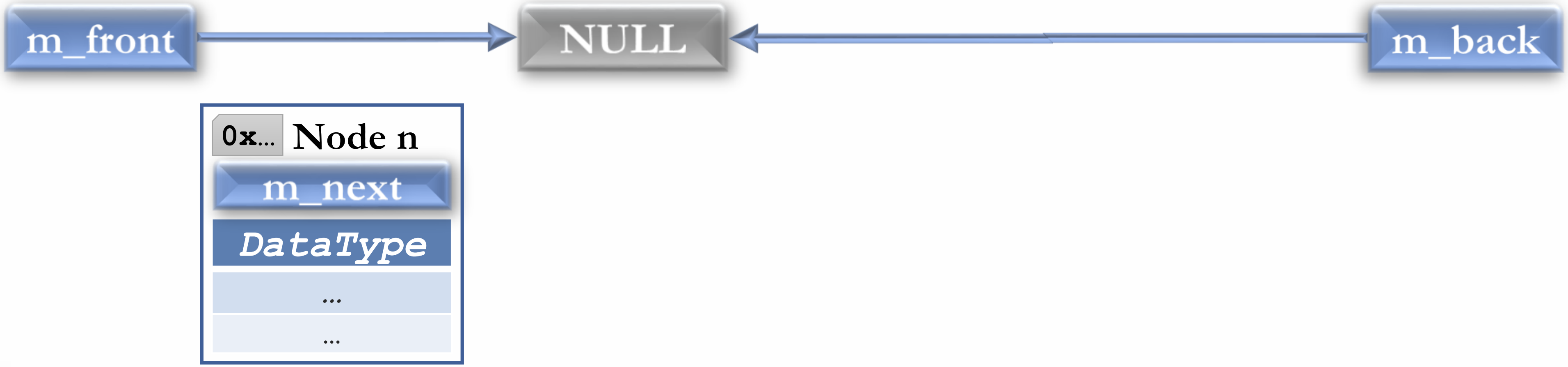
3) `m_back = newNode_Pt;`

List-based Queue(s)

List-based Implementation(s)

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

An originally empty Queue.

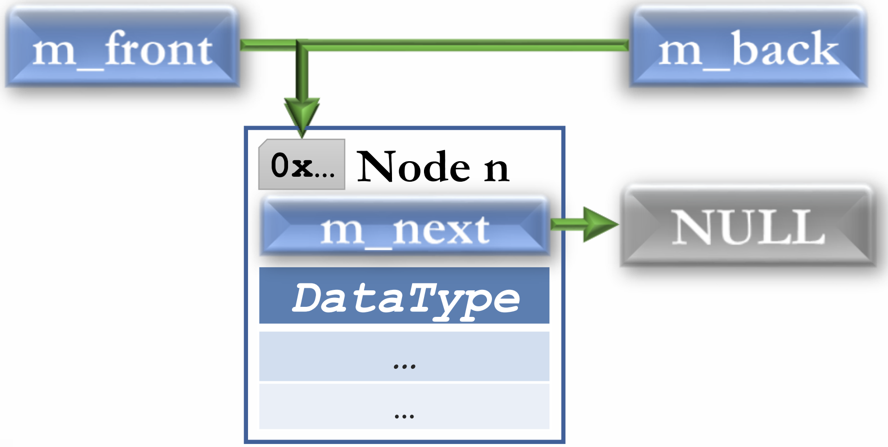


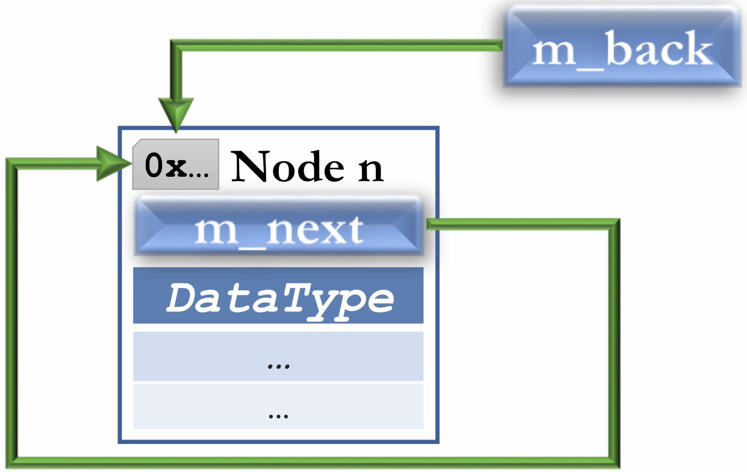
List-based Queue(s)

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;`

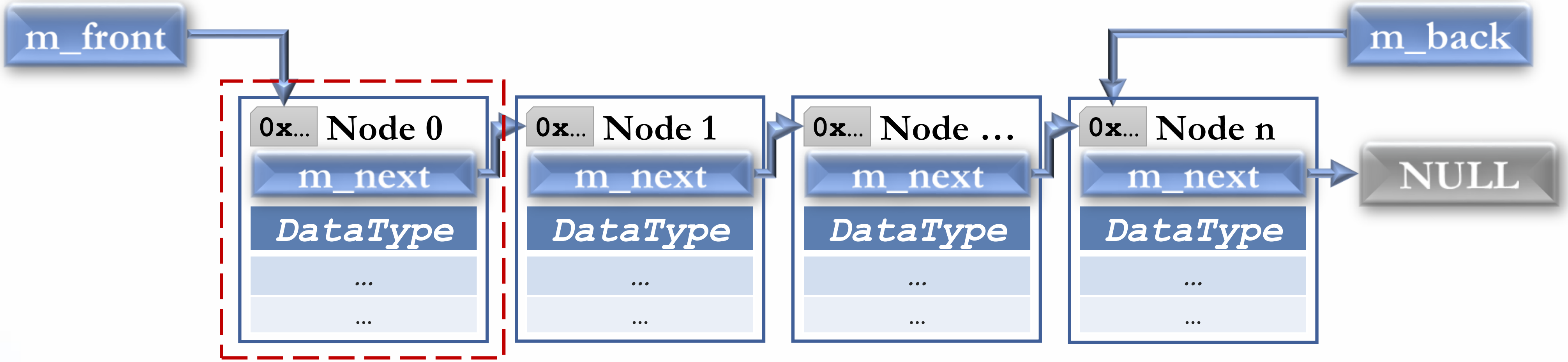
1) `newNode_Pt->m_next = newNode_Pt;`
2) `m_back = newNode_Pt;`

List-based Queue(s)

List-based Implementation(s)

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

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

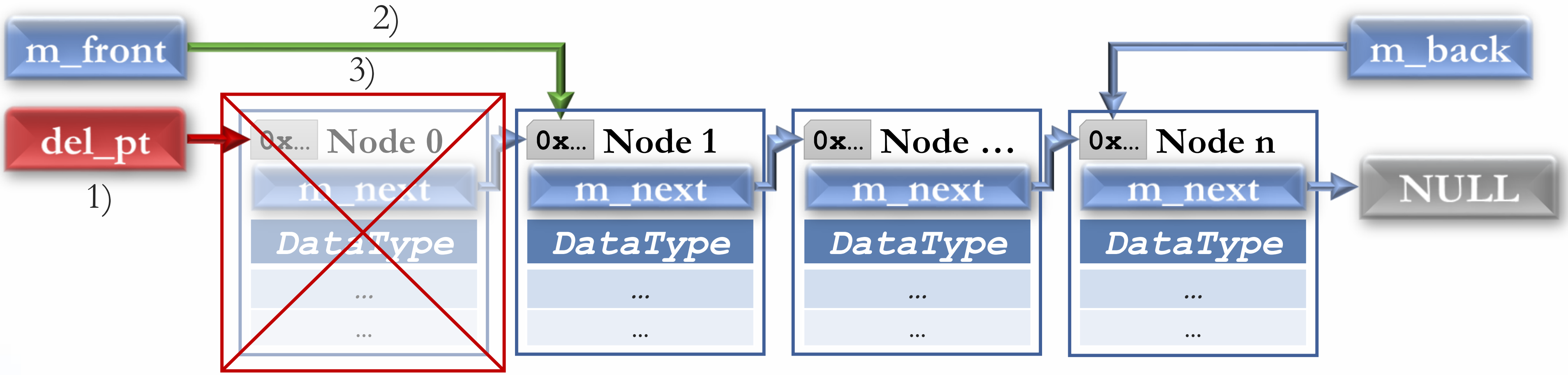


List-based Queue(s)

List-based Implementation(s)

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

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



1) `Node * del_pt = m_front;`

2) `m_front = m_front->m_next;`

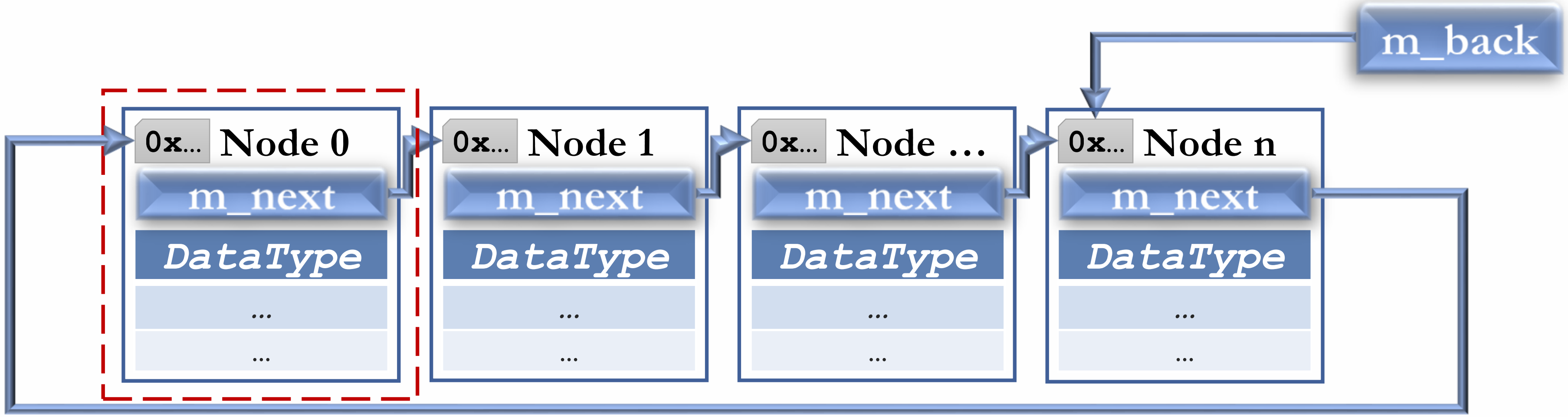
3) `delete del_pt;`

List-based Queue(s)

List-based Implementation(s) – *Unusual Case*

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

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

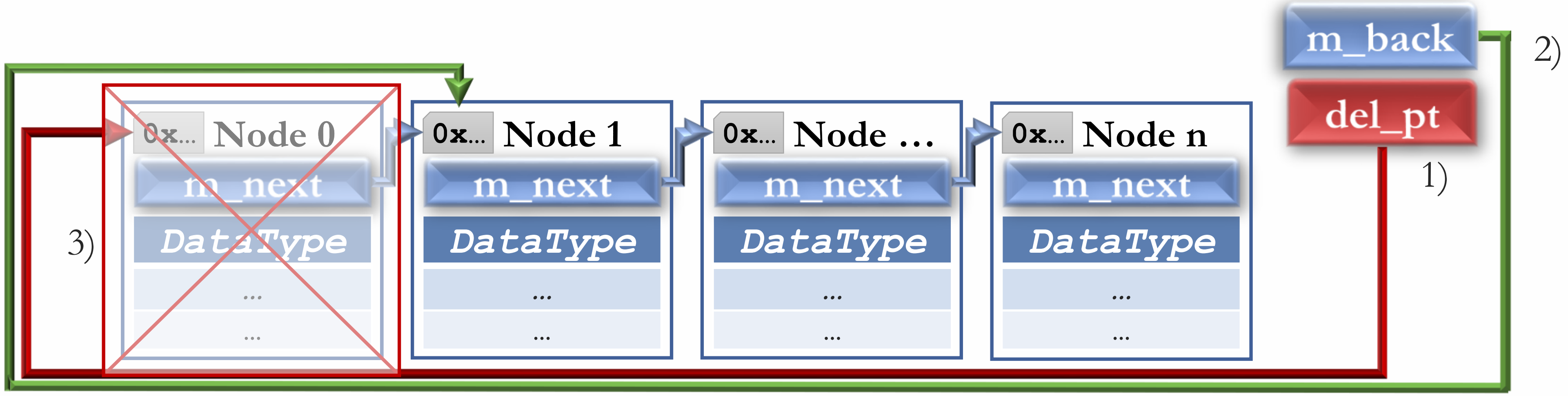


List-based Queue(s)

List-based Implementation(s) – *Unusual Case*

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 = del_pt->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;
    size_t size() const;
    void push(const DataType & value);
    void pop();
    void clear();
    DataType & front();
    DataType & back();
  private:
    Node * m_front, * m_back;
    size_t m_size;
};
```

```
class Node {
 friend class Queue;
 public:
 Node()
  : m_next(NULL) { }
 Node(const DataType & data, Node * next = NULL)
  : m_next(next), m_data(data) { }
 Node(const Node & other)
  : m_next(other.m_next), m_data(other.m_data) { }
 DataType & data(){
   return m_data;
 }
 const DataType & data() const{
   return m_data;
 }
 private:
  Node * m_next;
  DataType m_data;
};
```

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(s)

Queue Applications

A Discrete–Event "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 arriving at the bank.
- ➢ Each customer's request has a separate required service time.
- ➢ Only the customer who is at the front of the queue can be served.
- ➢ This customer is followingly removed from the system.

CS-202

Time for Questions !