CMPT 225

Lecture 8 – Data collection **Queue** as an ADT Class

Learning Outcomes

- At the end of this lecture, a student will be able to:
  - Describe Queue
  - Define public interface of Queue ADT
  - Design and implement Queue ADT using various data structures
  - Compare and contrast these various implementations using Big O notation
  - Give examples of real-life applications (problems) where we could use Queue to solve the problem
  - Solve problems using Queue ADT

Last Lecture

- We can now …
  - Describe Stack
  - Solve a problem using a Stack ADT class
    - Design and implement a Stack ADT class
      - Define its Public Interface
      - Design (and draw) and implement Stack ADT using various data structures (CDTs)
      - Compare and contrast these various implementations using Big O notation
      - Give examples of real-life applications (problems) where we could use Stack to solve the problem
- *Perform complexity analysis and use the Big O notation to represent time and space efficiency*

But we did not get to that! 😐

Today's menu

- Introduce our next linear data collection -> **Queue**
- Perform complexity analysis and use the Big O notation to represent time and space efficiency

Queue

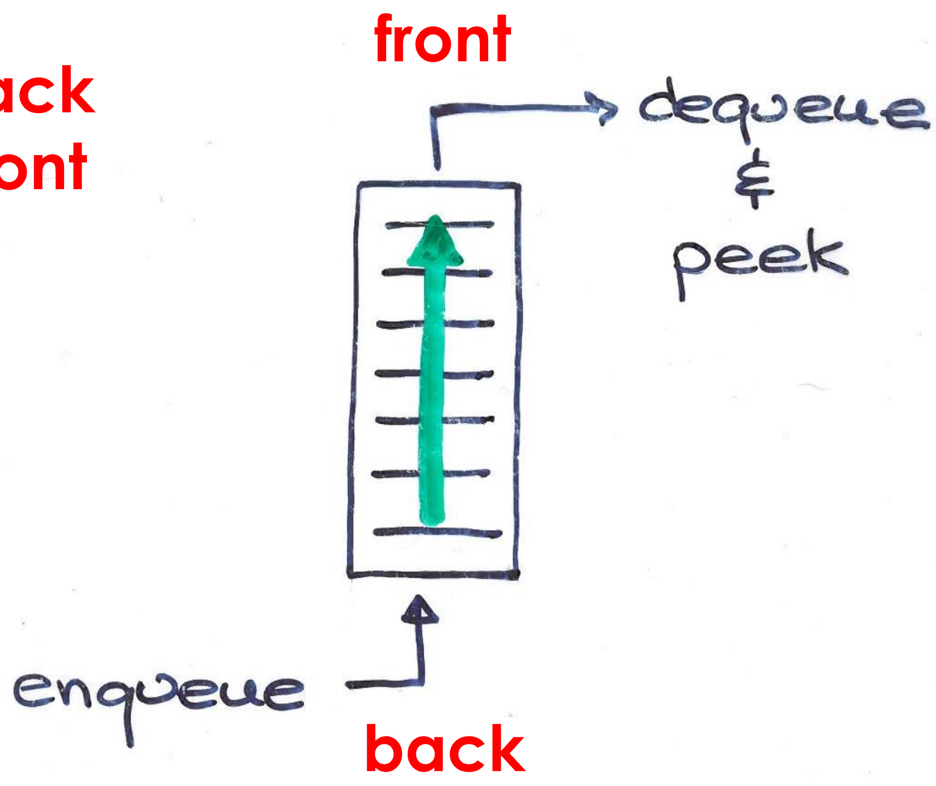
- What can we do with a Queue in the real world?
- How does a Queue behave in the real world?
  - Rule?
  - Duplications allowed?
  - Are people in a Queue sorted?
  - Must we keep track of the number of people in a Queue? Or must we simply ascertain whether the Queue is empty?

Source: https://compsci2014.wikispaces.com/file/view/queue_line_2.jpg/422130322/queue_line_2.jpg

What characterizes a Queue?

- A Queue only allows elements to be inserted at one end -> **back** and removed at the other -> **front**
- Access to other elements in a Queue is not *considered*
- **Rule**: FIFO / LILO
  - Fair: no starvation
    -> every element in the queue is processed
- Linear data collection
- **Not** a *general-purpose* (**not** a *flexible*) data collection

This rule becomes the Queue ***class invariant***.



Step 2 – Design - Queue operations

To become the **public interface** of our Queue ADT class

- **`enqueue`**: Insert element at back of Queue
- **`dequeue`**: Remove front element of Queue
- **`peek`**: Gives access to the front element of Queue (but does not remove it from the Queue)
- **`isEmpty`**: Is the Queue empty?

About Queue operations and **class invariant**

- It is the implementation of the methods of our Queue class that ensures that the **class invariant** (LILO or FIFO) holds true!
  - We write our code such that …
    - **`enqueue`** method **only** inserts an element at the **back** of the Queue
    - **`dequeue`** method **only** removes the element at the **front** of the Queue
    - **`peek`** method **only** peeks at the element at the **front** of the Queue
- Therefore it is important to clearly define and indicate where the **front** and the **back** of the Queue is located

Step 3 – Implementation – `Queue.h`

Queue public interface – Contract

NOTE: Expressed in C++ and using template and exceptions

Class invariant: FIFO / LILO

```
// Description: Returns true if this Queue is empty otherwise false.
// Postcondition:  This Queue is unchanged.
// Time Efficiency: O(1)
bool isEmpty( ) const;

// Description:  Adds a new element to the back of this Queue.
// Exception: Throws EnqueueFailedException if enqueue unsuccessful.
// Time Efficiency: O(1)
void enqueue(ElementType& newElement);
```

Step 3 – Implementation – `Queue.h`

Queue public interface – Contract

```
// Description: Removes the front element of this Queue.
// Precondition:  The Queue is not empty.
// Exception: Throws EmptyQueueException if this Queue is empty.
// Time Efficiency: O(1)
void dequeue( );
```

Alternative:

```
// Description: Removes and returns the front element of this Queue.
// Precondition:  The Queue is not empty.
// Exception: Throws EmptyQueueException if this Queue is empty.
// Time Efficiency: O(1)
ElementType & dequeue( );
```

Step 3 – Implementation – `Queue.h`

Queue public interface – Contract

```
// Description: Removes all elements from this Queue.
// Postcondition: Queue is in same state as when constructed.
```

~~// Precondition: The Queue is not empty.~~

~~// Exception: Throws EmptyQueueException if this Queue is empty.~~

```
void dequeueAll( );

// Description:  Returns (gives access) the front element of this Queue.
// Precondition:  The Queue is not empty.
// Postcondition:  This Queue is unchanged.
// Exceptions: Throws EmptyQueueException if this Queue is empty.
// Time Efficiency: O(1)
ElementType & peek( ) const;
```

Keep in mind that we want to perform these operations as fast as possible -> **O(1)**

Step 3 – Implementation of CDT underlying our Queue ADT class

1. **Array-based** implementation:

Private data members:

```
const unsigned int INITIAL_SIZE = 8;
unsigned int capacity = INITIAL_SIZE;
ElementType * elements = nullptr;
unsigned int front = 0;
unsigned int back = 0;
unsigned int elementCount = 0;
```

enqueue(…):
if FULL -> expand or error
elements[back] = newElement;
back++;
elementCount++;

dequeue( ):
if (isEmpty( )) -> error
front++;
elementCount−−;

index:	0	1	2	3	4	5
elements ->	3	7	5			
	front			**back**		

Keep in mind that we want to perform these operations as fast as possible -> **O(1)**

Step 3 – Implementation of CDT underlying our Queue ADT class

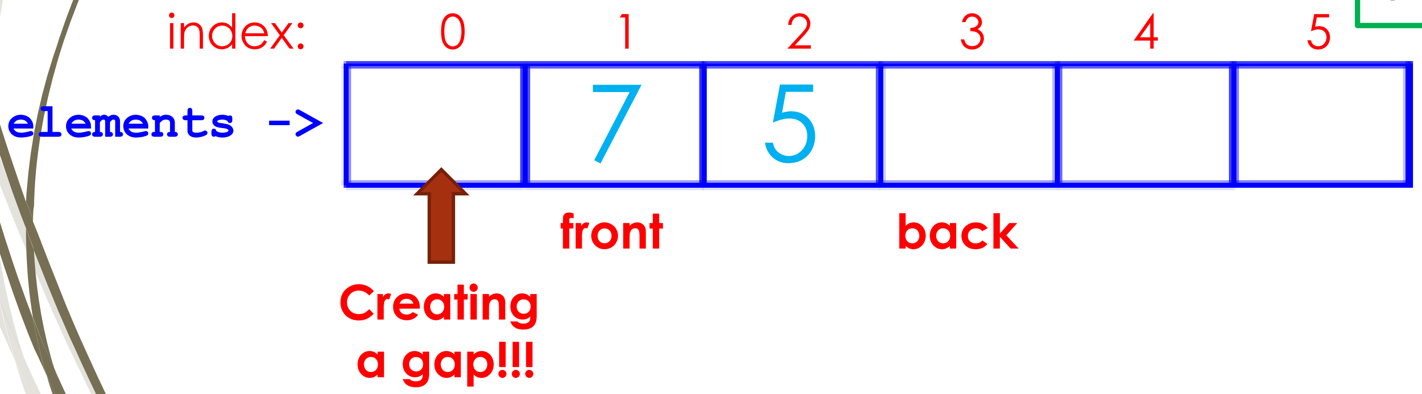
1. **Array-based** implementation:

Private data members:

```
const unsigned int INITIAL_SIZE = 8;
unsigned int capacity = INITIAL_SIZE;
ElementType * elements = nullptr;
unsigned int front = 0;
unsigned int back = 0;
unsigned int elementCount = 0;
```

enqueue(…):
if FULL -> expand or error
elements[back] = newElement;
back++;
elementCount++;

dequeue( ):
if (isEmpty( )) -> error
front++;
elementCount−−;



Keep in mind that we want to perform these operations as fast as possible -> **O(1)**

Step 3 – Implementation of CDT underlying our Queue ADT class

1. **Array-based** implementation – **Solution: Circular array**

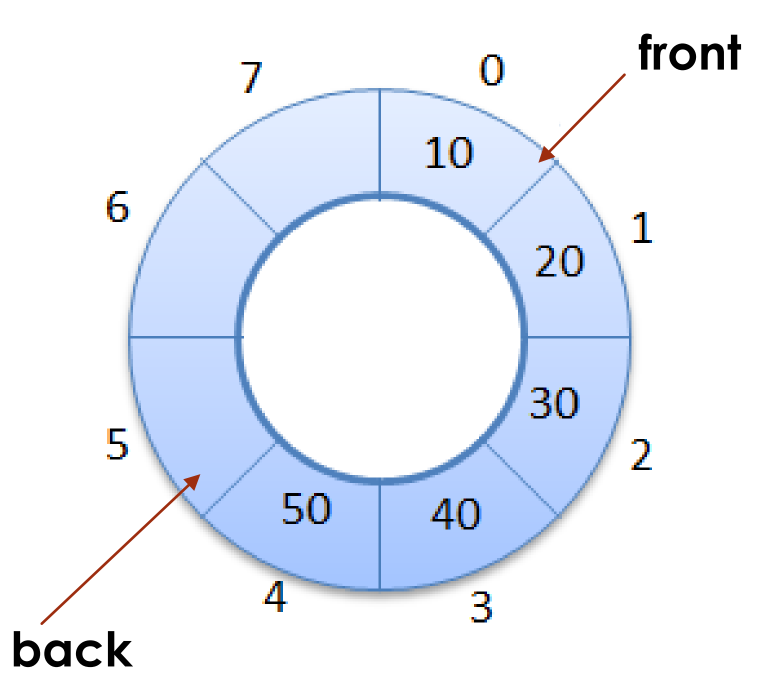
Private data members:

```
const unsigned int INITIAL_SIZE = 8;
unsigned int capacity = INITIAL_SIZE;
ElementType * elements = nullptr;
unsigned int front = 0;
unsigned int back = 0;
unsigned int elementCount = 0;
```

enqueue(...):
if FULL -> expand or error
elements[back] = newElement;
back = (back+1) % capacity; // current size
elementCount++;

dequeue( ):
if (isEmpty( )) -> error
front = (front + 1) % capacity; // current size
elementCount−−;

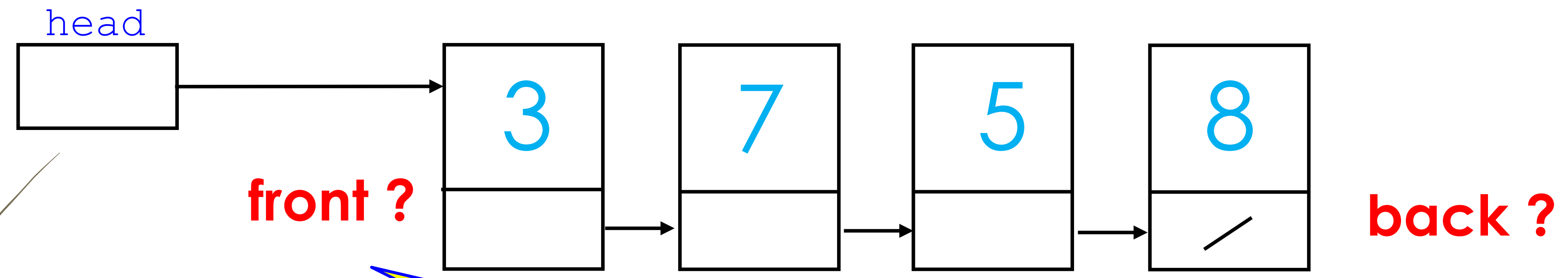
Represented here in a conceptual fashion



Source: https://www.oreilly.com/library/view/php-7-data/9781786463890/7bea3249-ea91-4d2e-9253-fca52296d5fc.xhtml

Step 3 – Implementation of CDT underlying our Queue ADT class

2. **Link-based** implementation



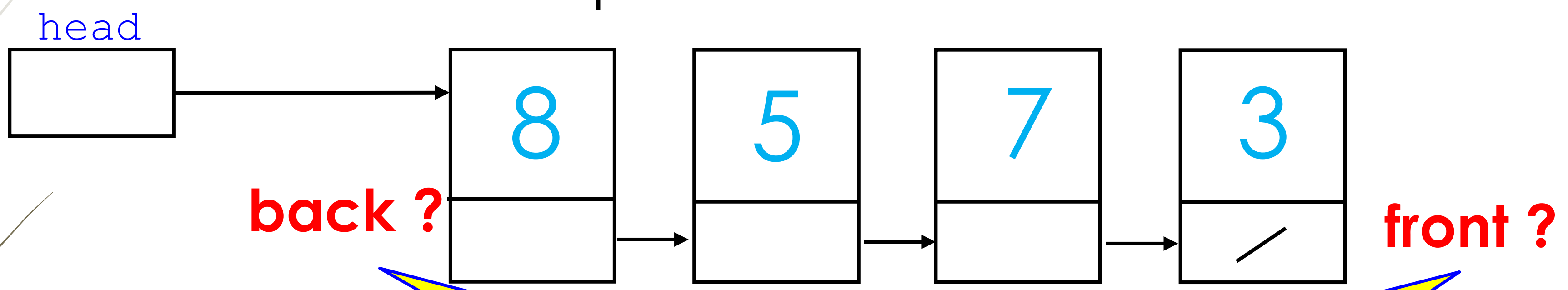
Private data member:

```
QueueNode * head;
unsigned int
  elementCount = 0;
```

Let's do some analysis!
Consider the Queue above: 3 7 5 8
What if the **front of the Queue** is at the **front of the linked list** and the **back of the Queue** is at the **back of the linked list**?
Would this allow us to implement the operations of our **Queue** class such that they execute in **O(1)**?

Step 3 – Implementation of CDT underlying our Queue ADT class

2. **Link-based** implementation



Private data member:
```
QueueNode * head;
unsigned int
  elementCount = 0;
```

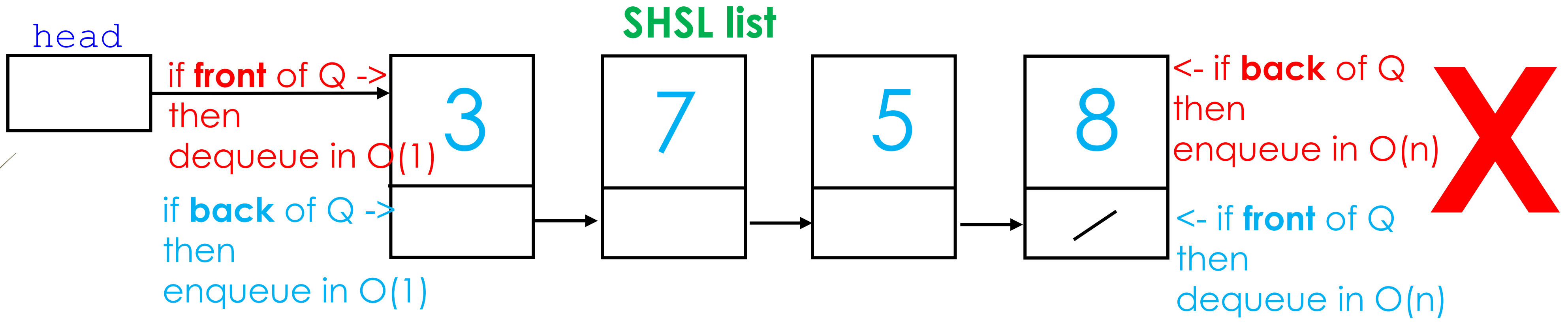
Let's do some more analysis!
If our Queue has not changed and is still: 3 7 5 8
Now, what if the **front of the Queue** is at the **back of the linked list** and the **back of the Queue** is at the **front of the linked list**?
Would this allow us to implement the operations of our **Queue** class such that they execute in **O(1)**?

Conclusion:

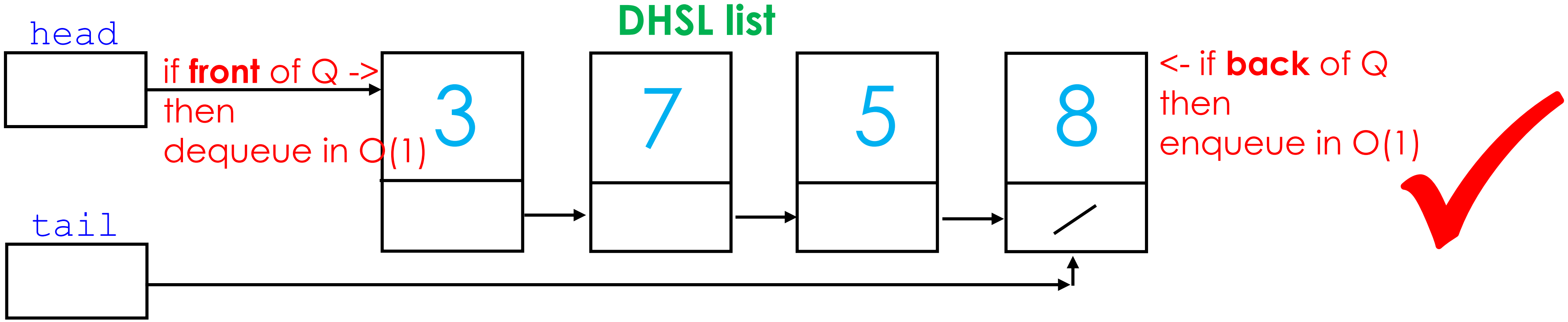
Step 3 – Implementation of CDT underlying our Queue ADT class

2. **Link-based** implementation

Private data member:
```
QueueNode * head;
unsigned int
 elementCount = 0;
```



Private data member:
```
QueueNode * head;
QueueNode * tail;
unsigned int
 elementCount = 0;
```



Step 3 - Implementation of Queue ADT class

3. List-based implementation

```
class Queue {
private:
   List * elements = nullptr;
public:  /* Queue public interface */
   bool isEmpty( ) const;
   bool enqueue(ElementType& newElement);
   bool dequeue( );
   bool dequeueAll( );
   ElementType & peek( ) const;
 };
```

Could we implement a **Queue** ADT class using a **List** ADT class?

Comparing various implementations of the Queue ADT class using Big O notation

- Time efficiency of Queue's operations (worst case scenario) expressed using the Big O notation

Operations	array-based	linked list-based	List-based
`isEmpty`			
`enqueue`			
`dequeue`			
`peek`			

When is a Queue appropriate?

- Examples of problem statements that can be solved using a Queue
  - **Pipeline architecture:** When module A's output is module B's input in a asynchronous fashion or when module B reads its input at a lower rate than module A produces its output, then a Queue can be used as a buffer
    - E.g.: Print queue, keyboard buffer
  - **Server requests**: Instant messaging servers queue up incoming messages
  - **Database requests**
  - **Operating systems** often use queues to schedule CPU jobs
  - **Event-driven software** like games enqueue events
    - Example of events: user presses a key on the on keyboard

Queue - Homework

- Assume you have a `myQueue` and `myStack` objects. Which elements would `myStack` contain once you have performed the following operations:

```
1. myStack.push(3)
2. myStack.push(6)
3. myStack.push(8)
4. myQueue.enqueue(myStack.peek())
5. myStack.push(5)
6. myQueue.enqueue(myStack.peek())
7. myStack.pop()
8. myStack.pop()
9. myQueue.enqueue(myStack.peek())
10.myStack.pop()
11.myStack.pop()
12.myStack.push(myQueue.peek())
```

√ Learning Check

- We can now …
  - Describe Queue
  - Define public interface of Queue ADT
  - Design and implement Queue ADT using various data structures (CDTs)
  - Compare and contrast these various implementations using Big O notation
  - Give examples of real-life applications (problems) where we could use Queue to solve the problem
  - Solve problems using Queue ADT

Next Lecture

- Review of simple **Sorting algorithms**