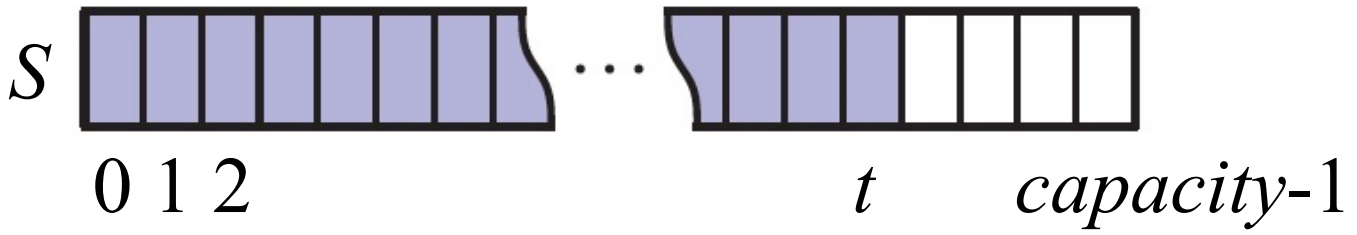
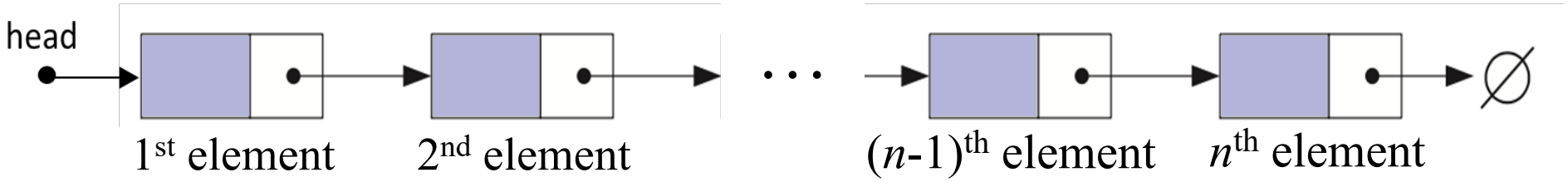
IMPLEMENTATION VIA A LINKED LIST

- Instead of implementing a stack using an array:



How about implementing it using a linked list:



THE STACK CLASS STRUCTURE

- Main Attributes:
  - A linked list to store the elements of the stack
  - A counter of the element in the stack

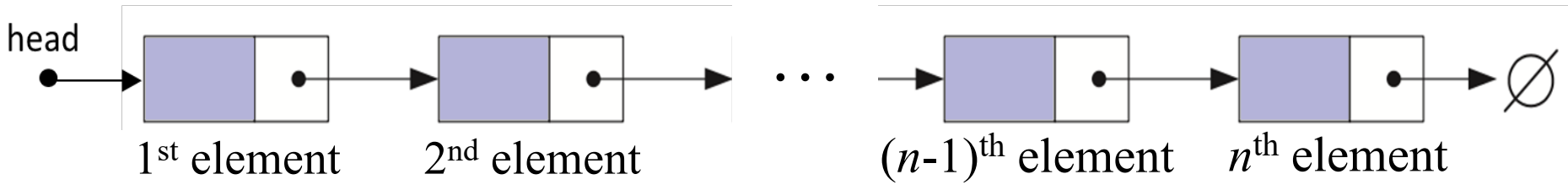
- Main Methods:
  - top(), push(e), pop(), size() , empty()

CLASS DEFINITION

Compared to the array-based implementation, this part is **exactly the same**, except that "push" now doesn't throw a "StackFull" exception, since the list size has no upper limit

Compared to the array-based implementation, "S" is now defined as a linked list instead of an array, and we now have "n" instead of "t" and "capacity"

```
typedef string Elem;
class LinkedStack{
public :
    LinkedStack(); // constructor
    int size() const ;
    bool empty() const ;
    const E& top() const throw( StackEmpty );
    void push(const E& e);
    void pop() throw( StackEmpty );
private:
    SLinkedList<Elem> S; // linked list
    int n; // number of elements in stack
};
```

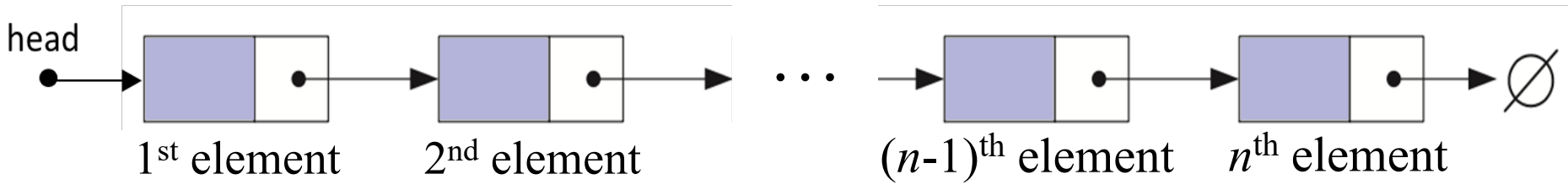


METHOD DEFINITION

```
LinkedStack::LinkedStack(): S(), n(0) { } // constructor, initializes S and sets n=0

int LinkedStack::size() const {// returns the number of elements
    return n;                                                    O(1)
}

bool LinkedStack::empty() const { // returns true if stack is empty
    return n == 0;                                               O(1)
}
const Elem& LinkedStack::top() const throw(StackEmpty) { // return top of stack
    if (empty()) throw StackEmpty("Top of empty stack");
    return S.front();                                            O(1)
}
```



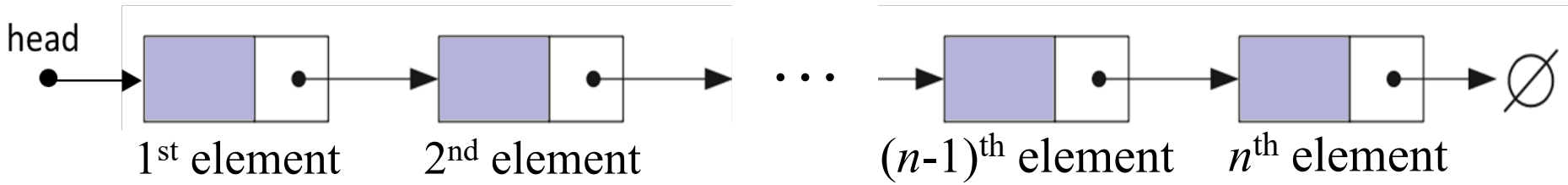
METHOD DEFINITION

```
void LinkedStack::push(const Elem& e) // push “e” onto stack
{
    ++n;
    S.addFront(e);
}

void LinkedStack::pop() throw(StackEmpty) // pop the stack
{
    if (empty()) throw StackEmpty("Pop from empty stack");
    --n;
    S.removeFront();
}
```

$O(1)$

$O(1)$



STACKS IN C++

- C++ provides a readily-available implementation of a stack (i.e., you do not have to implement it yourself).
- This comes as part of the "**Standard Template Library**" (STL) of C++, e.g., here is how to define a stack of integers:

```
#include <stack>
using std::stack;
stack<int> myStack;
```

This line defines `myStack` as a stack in which the elements are of type `int`; in this case we say the **base type** is `int`

ADT VS. STL

- **Abstract Data types (ADT)** is used to describe data structures without any implementation details
- **Standard Template Library (STL)**, which are readily-available data structures provided by C++

- **There might be some differences between the two:**
  - Example: In a stack **ADT**, we said earlier that the methods `pop()` and `top()` return **an error if the stack is empty**. While with stacks in C++'s **STL**, these two methods **do not throw an exception** if the stack is empty. Even though no exception is thrown, it may result in your program aborting!

STACK IMPLEMENTATION

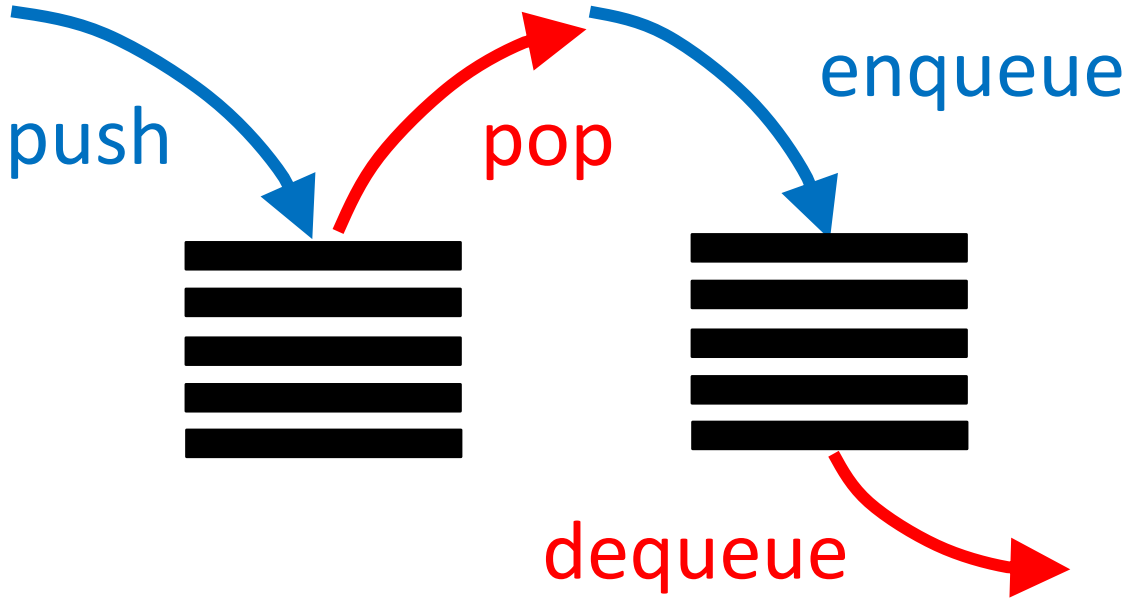
- Practically, whenever you need to use a data structure such as a stack, you may use the available implementation in C++'s STL
  - Nevertheless, to better understand how data structures work, we will be required to implement them from scratch.
  - Moreover, you may want to handle exceptional cases an STL does not!

QUEUES

QUEUES

- A **queue** is a data structure that stores objects which can be added or removed according to the "***First-in First-out (FIFO)***" principle.

Stack: Last in, first out **Queue:** First in, first out



QUEUES: ADT

- Formally, a queue is an abstract data type (ADT) that supports the following operations:
  - `enqueue(e):` **Insert element** `e` at the rear of the queue.
  - `dequeue():` **Remove element** at the front of the queue; an error occurs if the queue is empty.
  - `front():` Return a reference to the **value of the front element** in the queue; an error occurs if the queue is empty.
  - `size():` Return the **number of elements** in the queue.
  - `empty():` Return **`true` if the queue is empty** and false otherwise.

EXAMPLE

- In this example, if we write a queue as follows:

  (x,y,z)

  it means that:
  - z is the **back** of the queue
  - x is at the **front**

Now, let's fill this table!

P.S., under "**Output**" write:

- "error" if there is an error
- The value returned (if the function returns a value)
- "–" if there is **no error** and the function returns **no value**

Operation	*Output*	*front* ← *Q* ← *rear*
enqueue(5)	–	(5)
enqueue(3)	–	(5,3)
front()	5	(5,3)
size()	2	(5,3)
dequeue()	–	(3)
enqueue(7)	–	(3,7)
dequeue()	–	(7)
front()	7	(7)
dequeue()	–	()
dequeue()	"error"	()
empty()	true	()

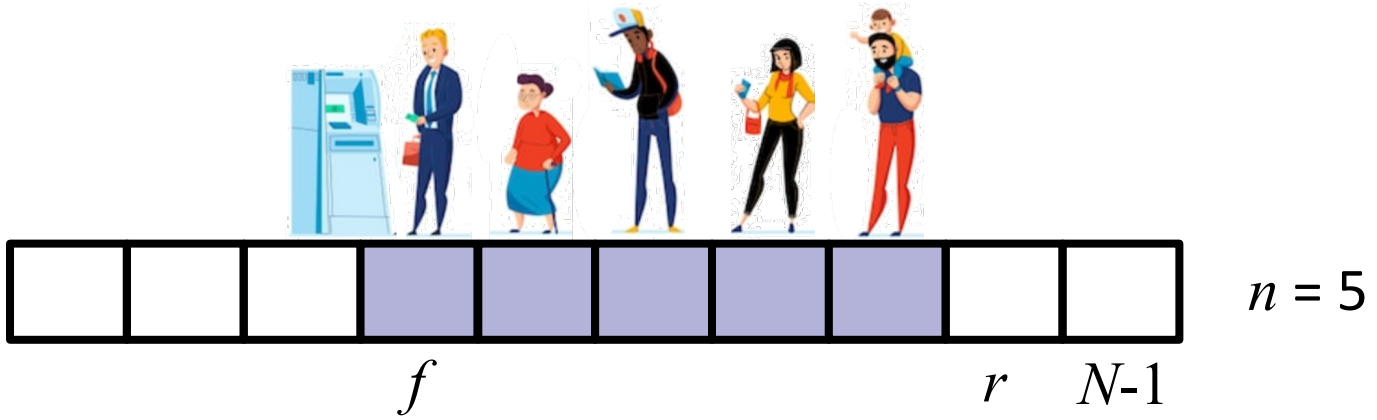
ARRAY-BASED QUEUE

- In the array-based implementation of this queue, it helps to **imagine the ATM machine moving forward**, rather than the customers!

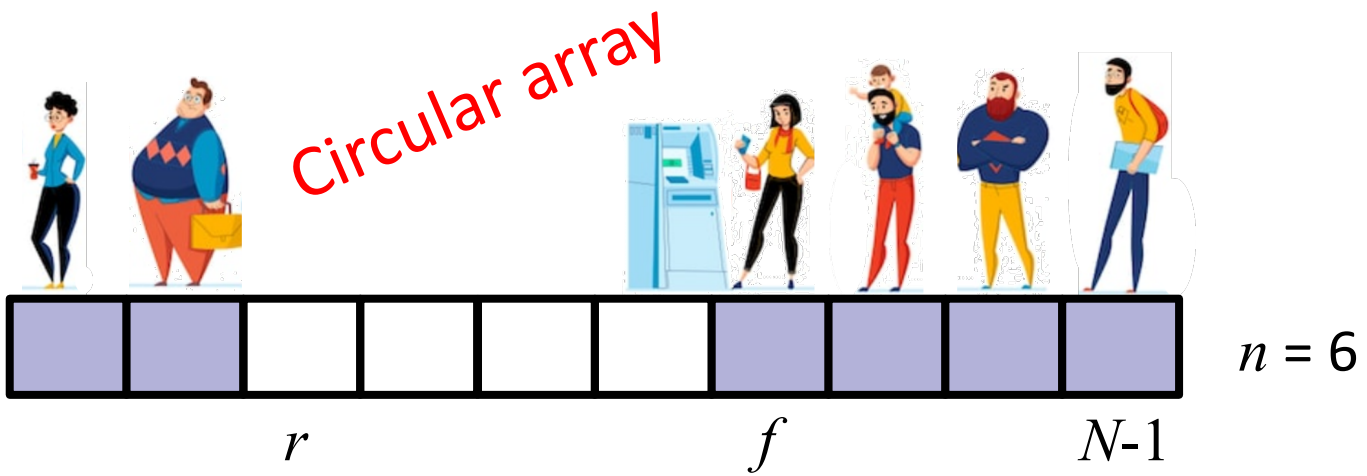
ARRAY-BASED QUEUE

f = index of front customer; r = index where a new joiner should stand;
n = No. of elements; N = array size (i.e. capacity)

A potential state of the queue:

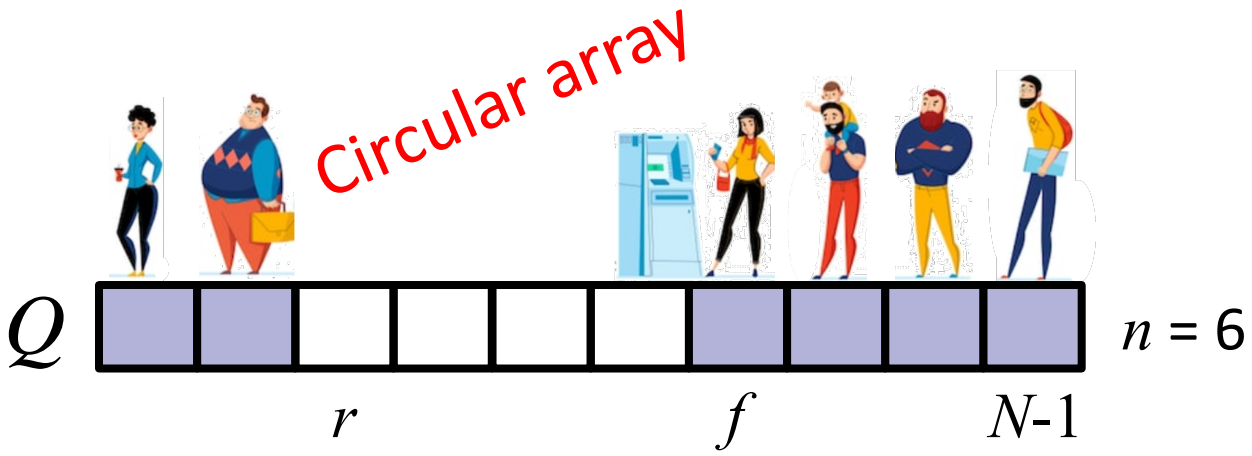


Three customers finished, and four joined the queue:



QUEUE PSEUDO CODE

Algorithm size():
  return n
Algorithm empty():
  return $(n = 0)$
Algorithm front():
  if empty() **then**
    throw QueueEmpty exception
  return $Q[f]$
Algorithm enqueue(e):
  if size() = N **then**
    throw QueueFull exception
  $Q[r] \leftarrow e$
  $r \leftarrow (r+1) \bmod N$
  $n \leftarrow n+1$
Algorithm dequeue():
  if empty() **then**
    throw QueueEmpty exception
  $f \leftarrow (f+1) \bmod N$
  $n \leftarrow n-1$



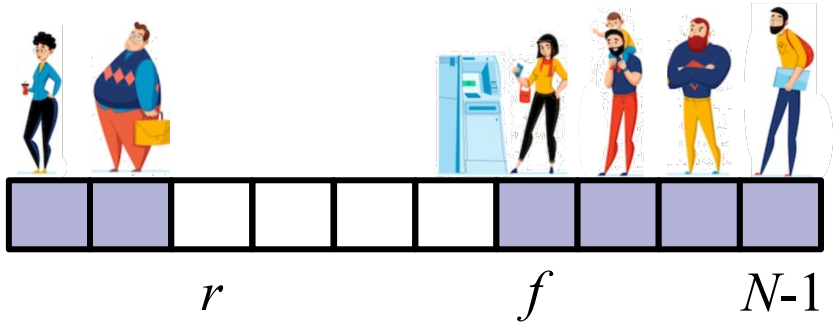
"(r+1) mod N" is the remainder after dividing (r+1) by N. That simply means:

"Increment r by 1, and if it falls out of the array, then go back to the beginning of the array."

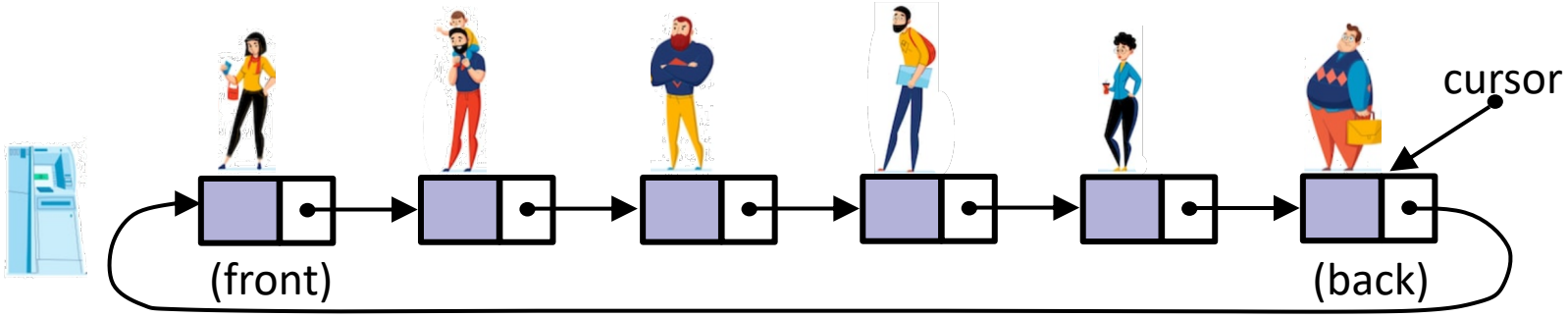
All methods are performed in O(1) time, while the space usage is O(N)

USING A CIRCULARLY-LINKED LIST

- Instead of implementing a queue using an “circular array”:



How about using a **circularly**-linked list instead?

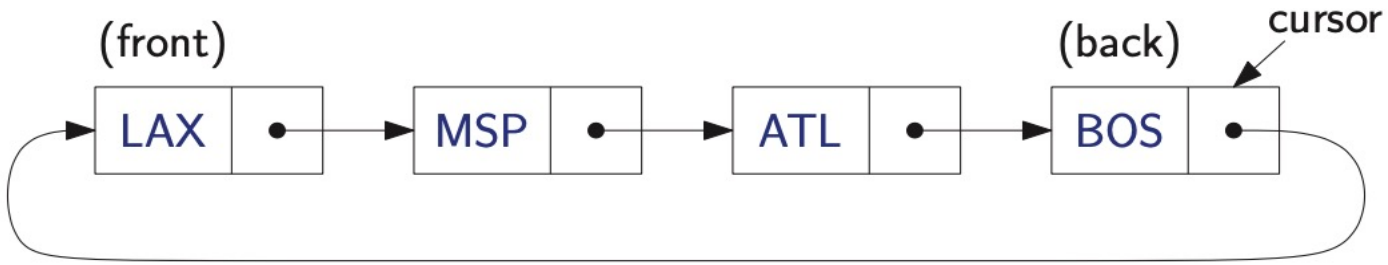


Although the list can be used to implement a queue that rotates (i.e., where the ATM moves forward, just like with arrays); the ATM will always be on the left!

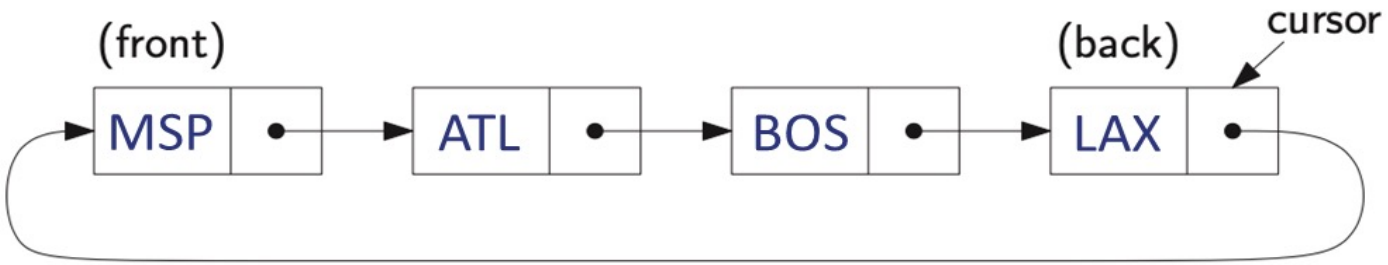
USING A CIRCULARLY-LINKED LIST

- Recall the main operations of a circularly-linked list:
  - `remove()`: **deletes the node at the front** of the list, i.e., the node right after the one that the curser points to.
  - `add(e)`: puts `e` in a new node, which is **added it to the front** of the list.
  - `advance()`: moves the **cursor one step forward** in the list

before calling "`advance()`":



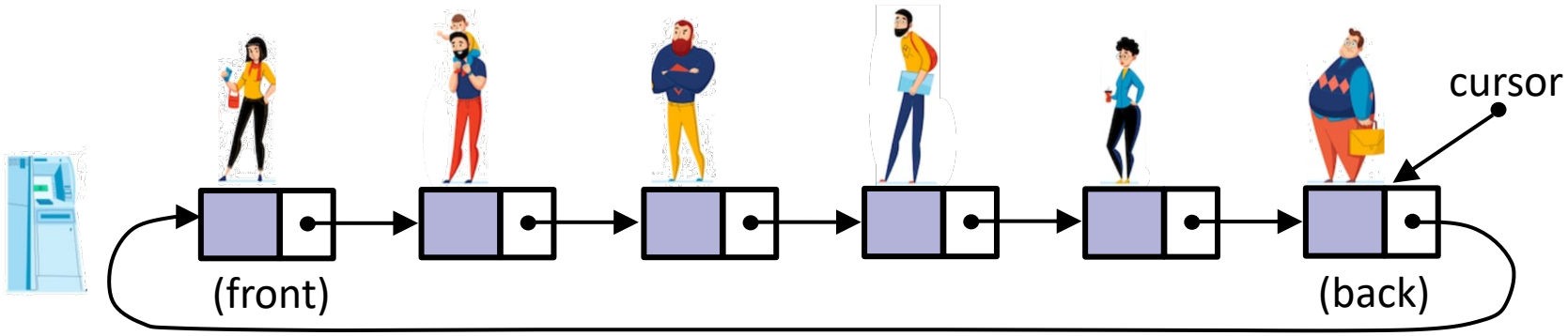
after calling "`advance()`":



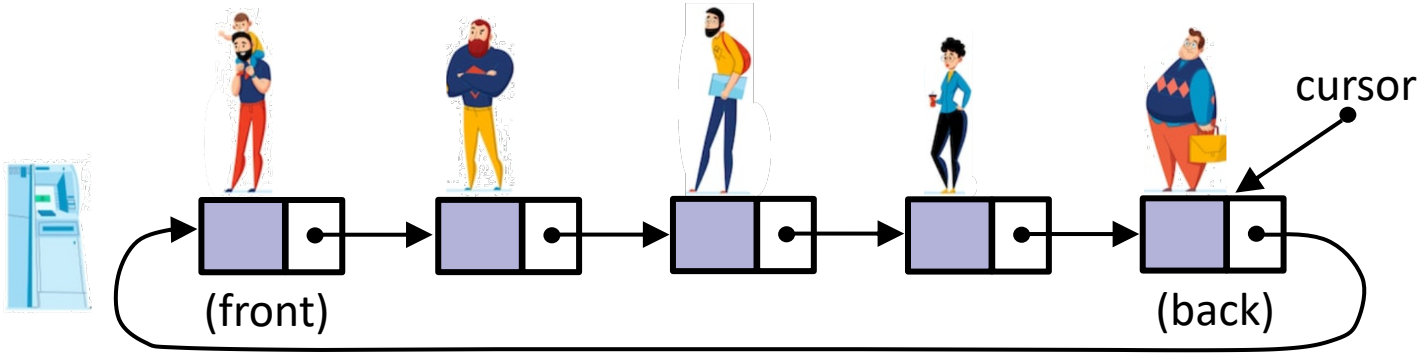
USING A CIRCULARLY-LINKED LIST

- To perform a **dequeue**, i.e., to remove the front customer:
  1. call the method `remove()`, which deletes the node at the front, i.e., the node right after the one that the curser points to

before calling "`remove()`":



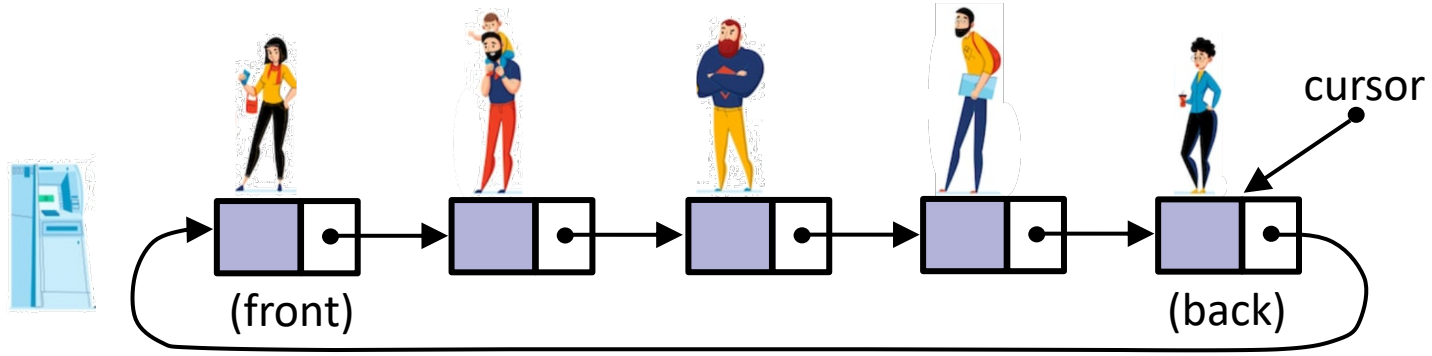
after calling "`remove()`":



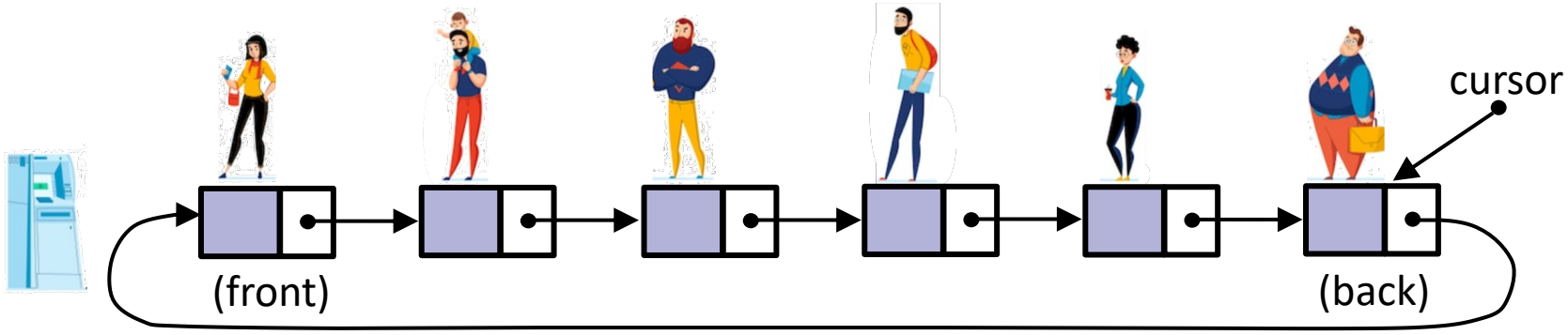
USING A CIRCULARLY-LINKED LIST

- To perform an **enqueue**, i.e., to add customer to the queue:
  1. call `add(e)`, which puts `e` in a **new node at the front** of the list
  2. then call `advance()`, which moves the **cursor one step forward**

before calling `add(e)` and `advance()`:

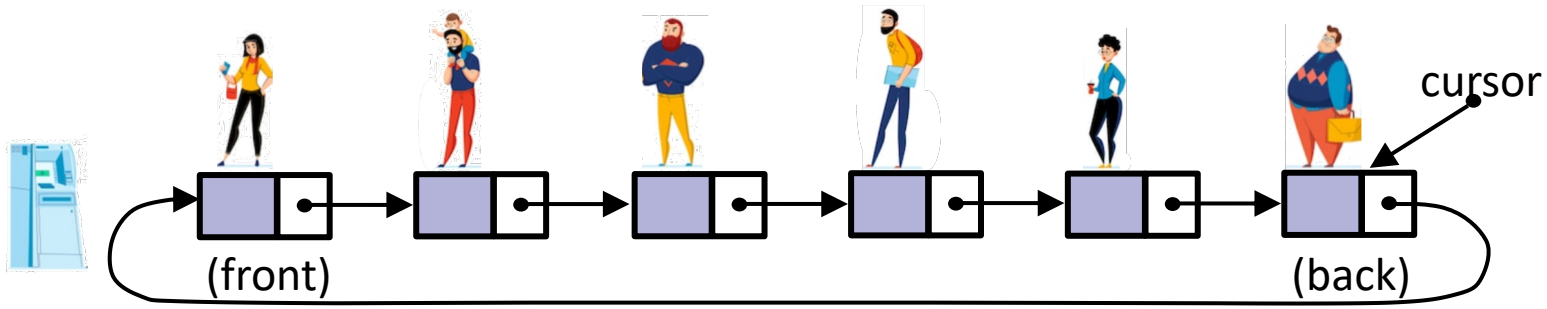


after calling `add(e)` and `advance()`:



C++ IMPLEMENTATION

```
typedef string Elem; // queue element type
class LinkedQueue {
    public:
        LinkedQueue(); // constructor
        int size() const; // number of items in the queue
        bool empty() const; // is the queue empty?
        const Elem& front() const throw(QueueEmpty); // the front element
        void enqueue(const Elem& e); // enqueue element at rear
        void dequeue() throw(QueueEmpty); // dequeue element at front
    private:
        CircleList C; // circular list of elements
        int n; // number of elements
};
```



C++ IMPLEMENTATION

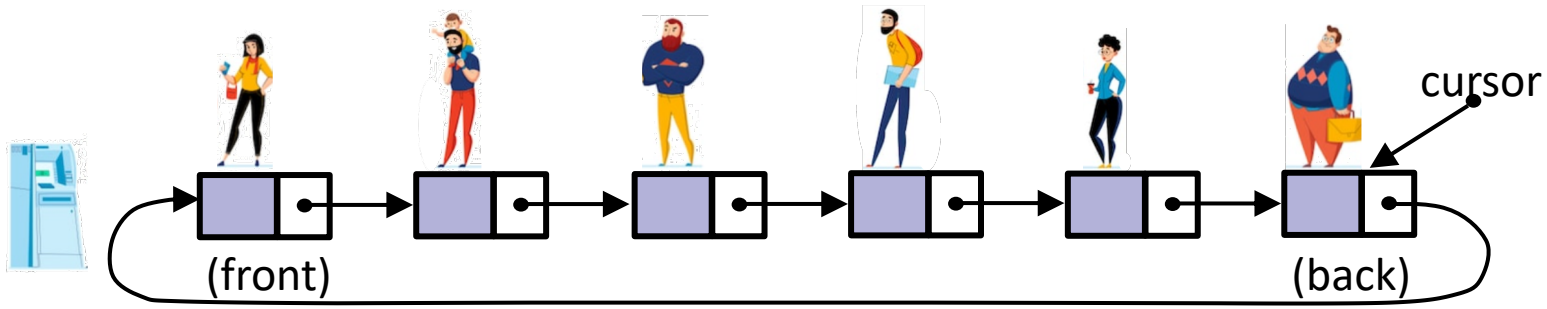
```
LinkedQueue::LinkedQueue() : C(), n(0) { } // constructor

int LinkedQueue::size() const { return n; } // number of items in the queue

bool LinkedQueue::empty() const { return n == 0; } // is the queue empty?

const Elem& LinkedQueue::front() const throw(QueueEmpty) {// get front element
    if (empty()) throw QueueEmpty("front of empty queue");
    return C.front(); }

void LinkedQueue::enqueue(const Elem& e) { // insert after cursor
    C.add(e);
    C.advance();
    n++; }
```



C++ IMPLEMENTATION

```
LinkedQueue::LinkedQueue() : C(), n(0) { } // constructor

int LinkedQueue::size() const { return n; } // number of items in the queue

bool LinkedQueue::empty() const { return n == 0; } // is the queue empty?

const Elem& LinkedQueue::front() const throw(QueueEmpty) {// get front element
    if (empty()) throw QueueEmpty("front of empty queue");
    return C.front(); }

void LinkedQueue::enqueue(const Elem& e) { // insert after cursor
    C.add(e);
    C.advance();
    n++; }

void LinkedQueue::dequeue() throw(QueueEmpty) {
    if (empty()) throw QueueEmpty("dequeue of empty queue");
    C.remove();
    n--; }
```

All methods are performed in O(1) time, and the space usage is O(n)

QUEUES: STL

- C++ provides a readily-available implementation of a queue (i.e., you do not have to implement it yourself).
- This comes as part of the "**Standard Template Library**" (STL) of C++, e.g., here is how to define a queue of floats:

```
#include <queue>
using std::queue;
queue<float> myQueue;
```

This line defines `myQueue` as a queue in which the elements are of type `float`; in this case, the "**base type**" is `float`

ADT VS. STL

- Reminder: Don't be confused between the following:
  - **Abstract Data types (ADT)**, which the textbook uses to describe data structures
  - **Standard Template Library (STL)**, which has data structures implemented in C++

- As we said earlier, there might be subtle differences between the two. For example:
  - The methods `enqueue()` and `dequeue()` in the queue **ADT** have different names in C++'s **STL**, which are `push()` and `pop()`, respectively.
  - Compared to the **ADT**, the queues in C++'s **STL** provides an additional method called `back()`, which returns a reference to the **element at the queue's rear**

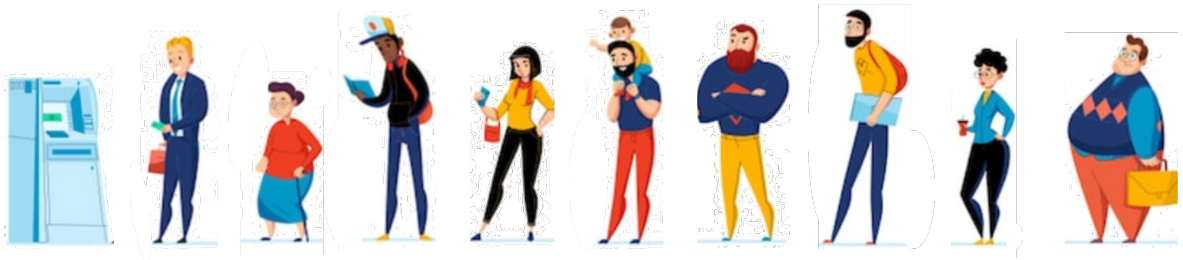
DEQUES

DOUBLE-ENDED QUEUE (DEQUE)

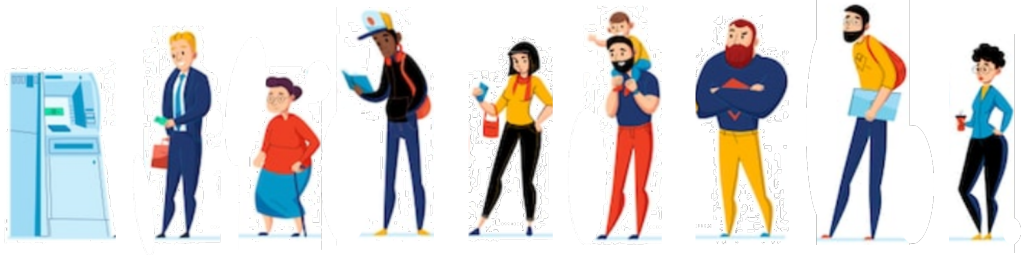
- It's a queue that supports **insertion** and **deletion** at both the **front** and the **back** of the queue!

A potential state of the queue:

A customer can skip the line!!



Last customer can now leave before reaching the ATM



DEQUE: ADT

- It's a queue that supports **insertion** and **deletion** at both the **front** and the **back** of the queue!
- Methods supported by DQ ADT:
  - `insertFront(e)`
  - `insertBack(e)`
  - `eraseFront()`
  - `eraseBack()`
  - `front()`
  - `back()`

Operation	*Output*	*D*
insertFront(3)	–	(3)
insertFront(5)	–	(5,3)
front()	*5*	(5,3)
eraseFront()	–	(3)
insertBack(7)	–	(3,7)
back()	7	(3,7)
eraseFront()	–	(7)
eraseBack()	–	()

DEQUES: STL

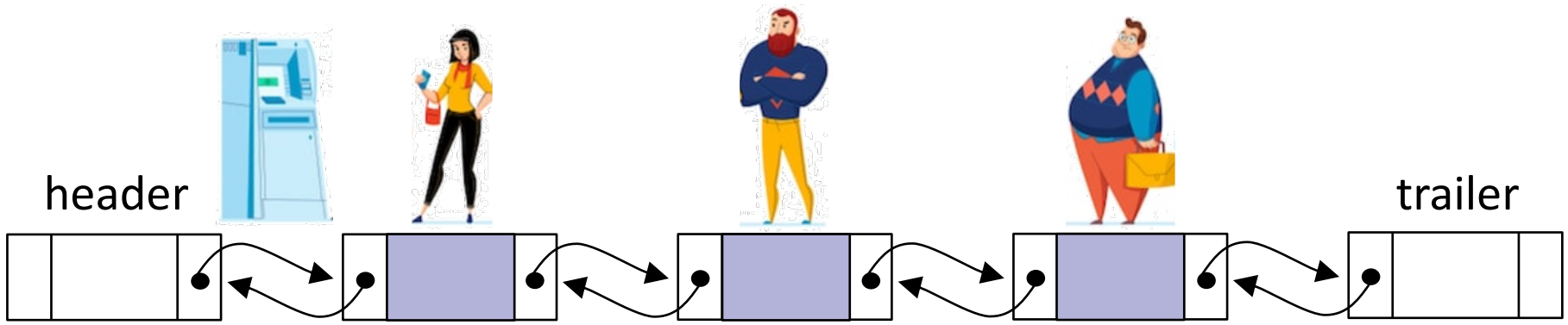
- C++ provides a readily-available implementation of a deque (i.e., you do not have to implement it yourself).
- This comes as part of the "**Standard Template Library**" (STL) of C++, e.g., here is how to define a queue of floats:

```
#include <deque>
using std::deque;
deque<string> myDeque;
```

This line defines deque as a deque in which the elements are of type string; in this case, the "**base type**" is string

USING A DOUBLY-LINKED LIST

- We will implement a deque using a **doubly**-linked list:



C++ IMPLEMENTATION

```
typedef string Elem; // deque element type
class LinkedDeque { // deque as doubly linked list
public:
    LinkedDeque(); // constructor
    int size() const; // number of items in the deque
    bool empty() const; // is the deque empty?
    const Elem& front() const throw(DequeEmpty); // the first element
    const Elem& back() const throw(DequeEmpty); // the last element
    void insertFront(const Elem& e); // insert new first element
    void insertBack(const Elem& e); // insert new last element
    void eraseFront() throw(DequeEmpty); // remove first element
    void eraseBack() throw(DequeEmpty); // remove last element
private: // member data
    DLinkedList D; // linked list of elements
    int n; // number of elements
};
```

C++ IMPLEMENTATION

```
void LinkedDeque::insertFront(const Elem& e) { D.addFront(e); n++; }

void LinkedDeque::insertBack(const Elem& e) { D.addBack(e); n++; }

void LinkedDeque::eraseFront() throw(DequeEmpty) {
    if (empty())  throw DequeEmpty("removeFront of empty deque");
    D.removeFront();
    n--;
}

void LinkedDeque::eraseBack() throw(DequeEmpty) {
    if (empty()) throw DequeEmpty("removeBack of empty deque");
    D.removeBack();
    n--;
}
```

All methods are performed in O(1) time, and the space usage is O(n)