Data Structures (in C++)

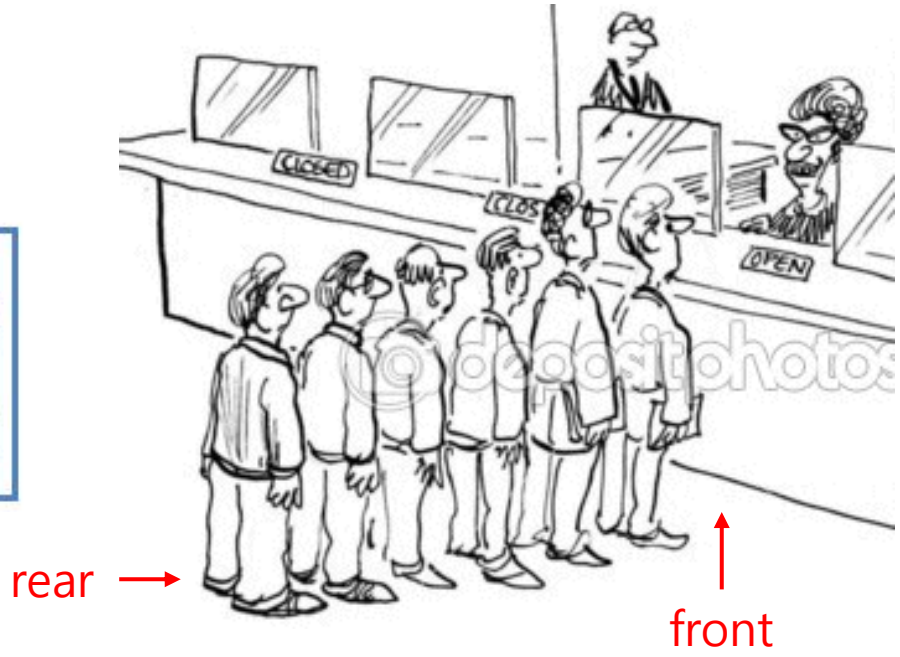
- Queues -

부산대학교 정보·의생명공학대학
정보컴퓨터공학부

Queues

Queues

- **Queue**
  - A container of objects that are inserted and removed according to the **first-in first-out (FIFO)** principle
  - Only the element that has been in the queue the longest can be removed
  - Elements enter the queue at the rear
  - Elements are removed from the front



Queue ADT

- The queue ADT 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, but do not remove, a reference to the front element in the queue; an error occurs if the queue is empty.

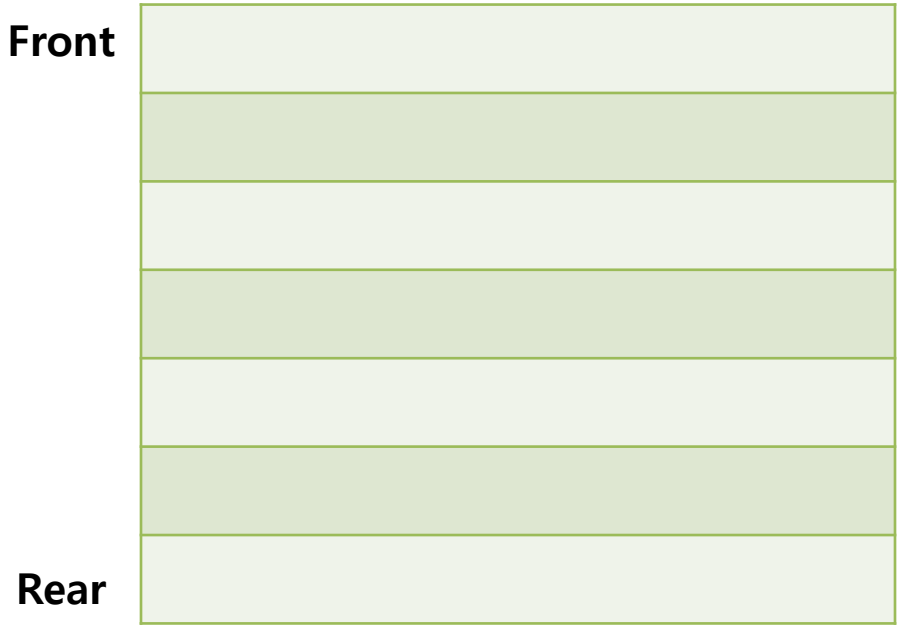
- Additional utility functions:

size(): Return the number of elements in the queue.

empty(): Return true if the queue is empty and false otherwise.

Queue Example

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*	()



The STL Queue

- The STL queue implementation is based on the STL deque or list class

```
#include <queue>
using std::queue;                    // make queue accessible
queue<float> myQueue;                // a queue of floats
```

size(): Return the number of elements in the queue.

empty(): Return true if the queue is empty and false otherwise.

push(*e*): Enqueue *e* at the rear of the queue.

pop(): Dequeue the element at the front of the queue.

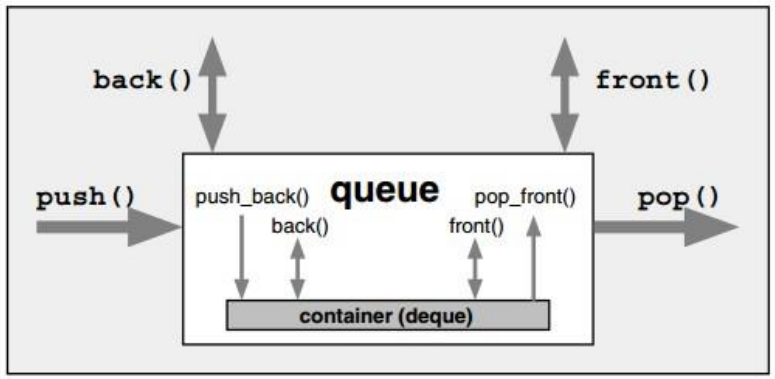
front(): Return a reference to the element at the queue's front.

back(): Return a reference to the element at the queue's rear.

- Applying *front()*, *back()*, or *pop()* to an empty queue is undefined

The STL Queue

- **The STL Queue Reference Manual**



std::queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

The std::queue class is a container adaptor that gives the programmer the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The queue pushes the elements on the back of the underlying container and pops them from the front.

Template parameters

T - The type of the stored elements. The behavior is undefined if T is not the same type as Container::value_type.

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of *SequenceContainer*. Additionally, it must provide the following functions with the usual semantics:

- back()
- front()
- push_back()
- pop_front()

The standard containers std::deque and std::list satisfy these requirements.

https://en.cppreference.com/w/cpp/container/queue

The STL Queue

- **The STL Queue Reference Manual**

Member functions

(constructor)	constructs the queue (public member function)
(destructor)	destructs the queue (public member function)
operator=	assigns values to the container adaptor (public member function)

Element access

front	access the first element (public member function)
back	access the last element (public member function)

Capacity

empty	checks whether the underlying container is empty (public member function)
size	returns the number of elements (public member function)

Modifiers

push	inserts element at the end (public member function)
push_range (C++23)	inserts a range of elements at the end (public member function)
emplace (C++11)	constructs element in-place at the end (public member function)
pop	removes the first element (public member function)
swap (C++11)	swaps the contents (public member function)

std::queue<T,Container>::push

```
void push( const value_type& value );
void push( value_type&& value );        (since C++11)
```

Pushes the given element `value` to the end of the queue.

1) Effectively calls `c.push_back(value)`
2) Effectively calls `c.push_back(std::move(value))`

std::queue<T,Container>::emplace

```
template< class... Args >
void emplace( Args&&... args );                    (since C++11) (until C++17)

template< class... Args >
decltype(auto) emplace( Args&&... args );          (since C++17)
```

Pushes a new element to the end of the queue. The element is constructed in-place, i.e. no copy or move operations are performed. The constructor of the element is called with exactly the same arguments as supplied to the function.

Effectively calls `c.emplace_back(std::forward<Args>(args)...);`

Parameters

args - arguments to forward to the constructor of the element

https://en.cppreference.com/w/cpp/container/queue

C++ Queue Interface

- **An Informal Queue Interface**

```
template <typename E>
class Queue {                                   // an interface for a queue
public:
  int size() const;                             // number of items in queue
  bool empty() const;                           // is the queue empty?
  const E& front() const throw(QueueEmpty);     // the front element
  void enqueue (const E& e);                    // enqueue element at rear
  void dequeue() throw(QueueEmpty);             // dequeue element at front
};
```

accessors

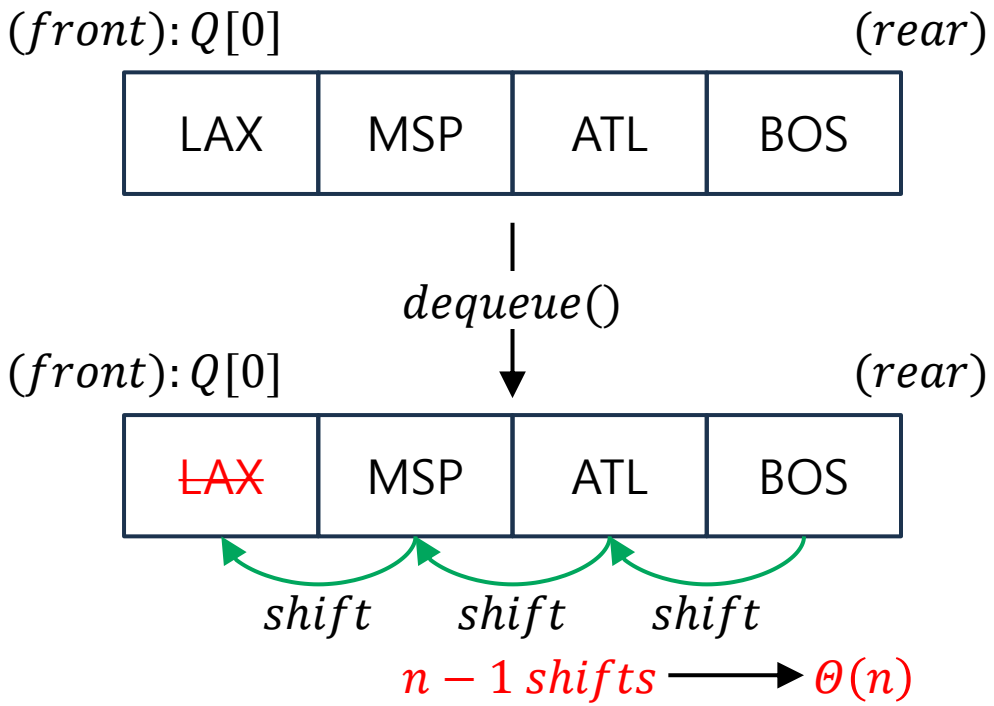
no return for dequeue

```
class QueueEmpty : public RuntimeException {
public:
  QueueEmpty(const string& err) : RuntimeException(err) { }
};
```

error message

Queue Implementation: Array-Based

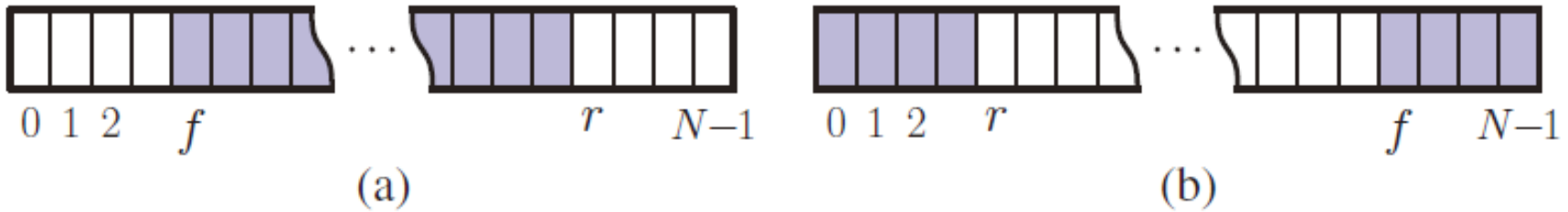
- **A Simple Array-Based Implementation**
  - The queue consists of an *N*-element array *Q*
  - *Q*[0] denotes the front
  - The dequeue operation is inefficient
    - Each operation shifts all of the remaining elements to the left



Queue Implementation: Array-Based

- **Using an Array in a Circular Way**
  - Q becomes a circular array
  - Indices wrap around the end of Q

- f is the index of the cell of Q storing the front of the queue. If the queue is nonempty, this is the index of the element to be removed by dequeue.
- r is an index of the cell of Q following the rear of the queue. If the queue is not full, this is the index where the element is inserted by enqueue.
- n is the current number of elements in the queue.



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

Queue Implementation: Array-Based

- **Modulo Operator**
  - Gives the remainder of an integer division
  - Given non-negative integers x and y,
    $x = qy + r$ where $0 \leq r < y$

```
+5 % +2 = +1,  +5 // +2 = +2
+5 % +5 = +0,  +5 // +5 = +1
-5 % +2 = -1,  -5 // +2 = -2
-5 % -5 = +0,  -5 // -5 = +1
+7 % +2 = +1,  +7 // +2 = +3
+7 % -2 = +1,  +7 // -2 = -3
-5 % -2 = -1,  -5 // -2 = +2
-7 % -3 = -1,  -7 // -3 = +2
```

- **From the C++ reference:**

The binary operator % yields the remainder of the integer division of the first operand by the second (after usual arithmetic conversions; note that the operand types must be integral types). If the quotient a/b is representable in the result type, `(a/b)*b + a%b == a`. If the second operand is zero, the behavior is undefined. If the quotient a/b is not representable in the result type, the behavior of both a/b and a%b is undefined (that means `INT_MIN%-1` is undefined on 2's complement systems)

Note: Until C++11, if one or both operands to binary operator % were negative, the sign of the remainder was implementation-defined, as it depends on the rounding direction of integer division. The function `std::div` provided well-defined behavior in that case.

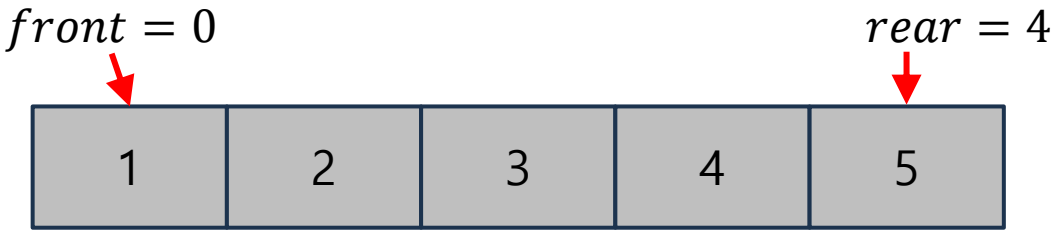
https://en.cppreference.com/w/cpp/language/operator_arithmetic

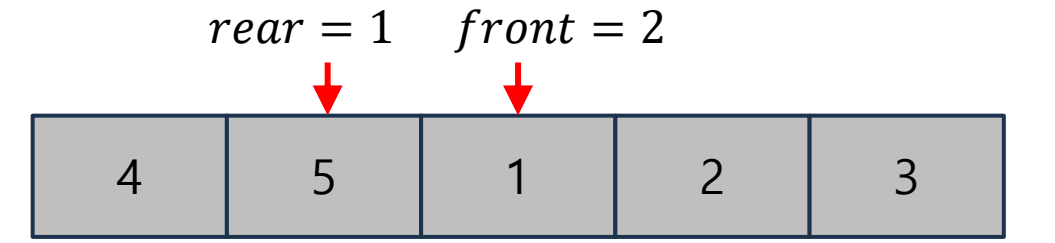
Queue Implementation: Array-Based

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

Queue Implementation: Array-Based

```
// Circular Queue implementation in C++
#include <iostream>
#define SIZE 5 /* Size of Circular Queue */

using namespace std;

class Queue
{
private:
    int items[SIZE], front, rear;
public:
    Queue()
    {
        front = -1;
        rear = -1;
    }
    // Check if the queue is full
    bool isFull()
    {
        if (front == 0 && rear == SIZE - 1)
        {
            // Edge case
            return true;
        }
        if (front == rear + 1)
        {
            return true;
        }
        return false;
    }
```

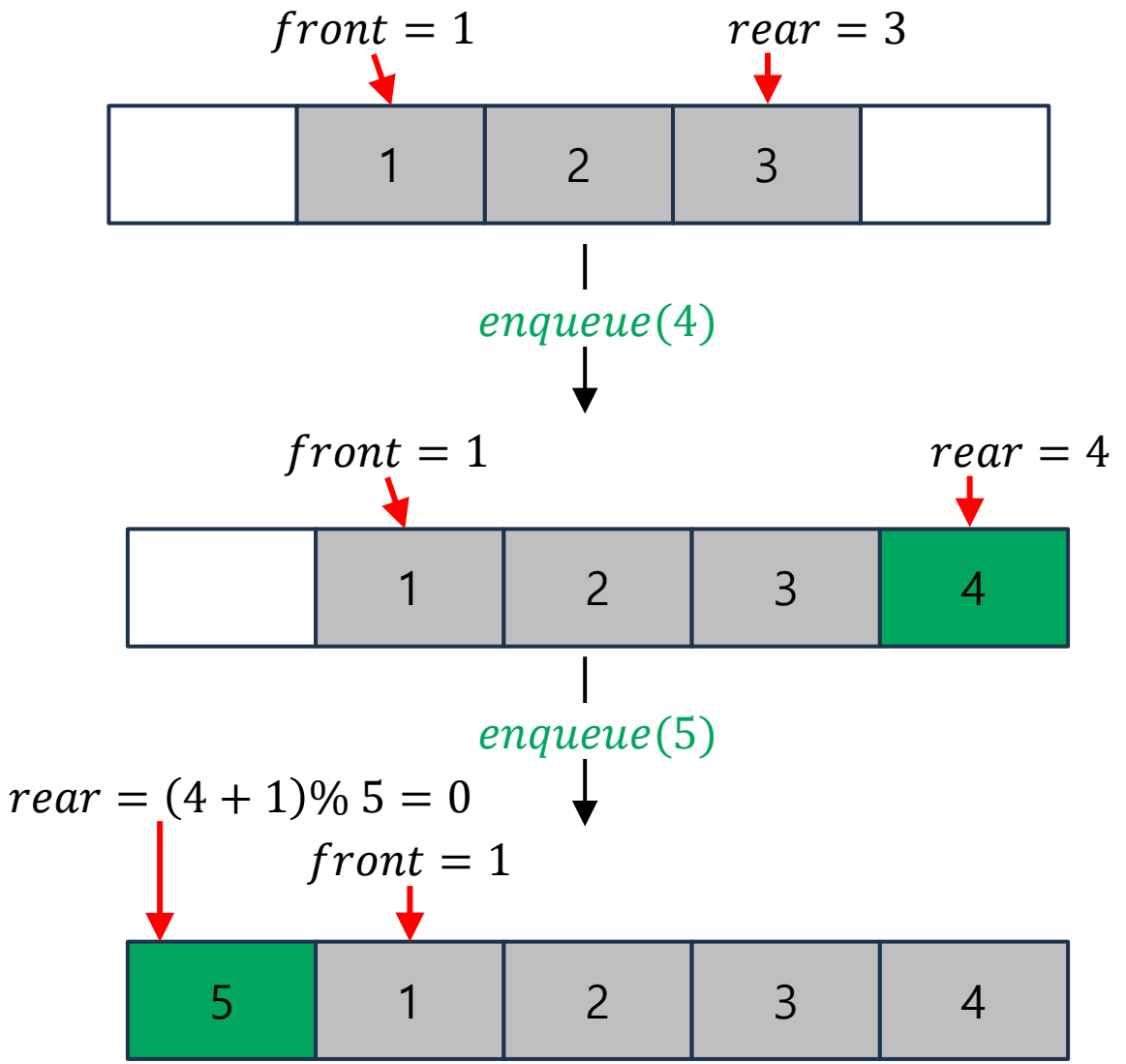




https://www.programiz.com/dsa/circular-queue

Queue Implementation: Array-Based

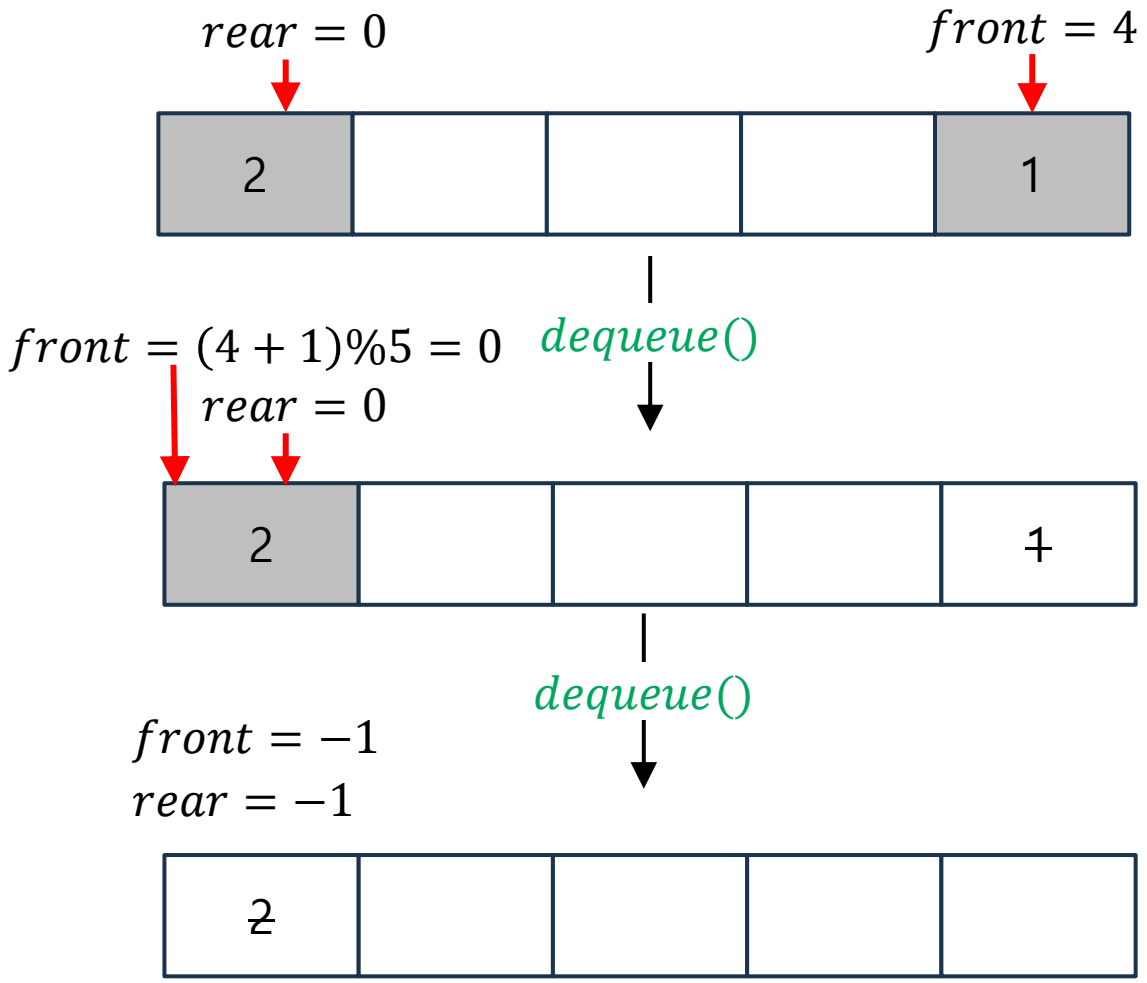
```
// Check if the queue is empty
bool isEmpty()
{
    if (front == -1)
        return true;
    else
        return false;
}

// Adding an element
void enQueue(int element)
{
    if (isFull())
    {
        cout << "Queue is full” << endl;
    }
    else
    {
        if (front == -1)
            front = 0;
        rear = (rear + 1) % SIZE;
        items[rear] = element;
        cout << endl
             << "Inserted " << element << endl;
    }
}
```



https://www.programiz.com/dsa/circular-queue

Queue Implementation: Array-Based

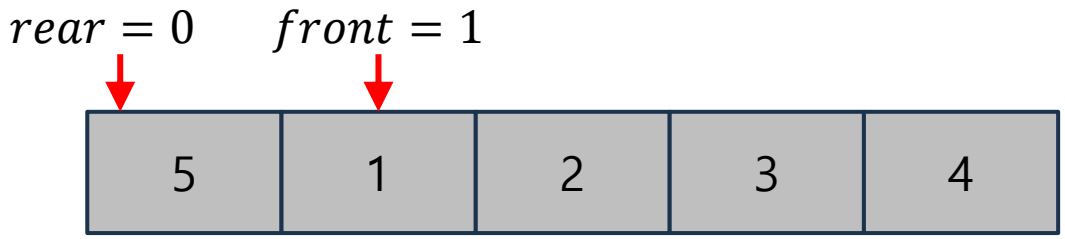
```
// Removing an element
int deQueue()
{
    int element;
    if (isEmpty())
    {
        cout << "Queue is empty" << endl;
        return (-1);
    }
    else
    {
        element = items[front];
        if (front == rear)
        {
            front = -1;
            rear = -1;
        }
        // Q has only one element,
        // so we reset the queue after deleting it.
        else
        {
            front = (front + 1) % SIZE;
        }
        return (element);
    }
}
```



https://www.programiz.com/dsa/circular-queue

Queue Implementation: Array-Based

```
    void display()
    {
        // Function to display status of Circular Queue
        int i;
        if (isEmpty())
        {
            cout << endl
                 << "Empty Queue" << endl;
        }
        else
        {
            cout << "Front -> " << front;
            cout << endl
                 << "Items -> ";
            for (i = front; i != rear; i = (i + 1) % SIZE)
                cout << items[i];
            cout << items[i];
            cout << endl
                 << "Rear -> " << rear << endl;
        }
    }
};
```




```
Front -> 1
Items -> 12345
Rear -> 0
```

https://www.programiz.com/dsa/circular-queue

Queue Implementation: Array-Based

```
int main()
{
    Queue q;
    // Fails because front = -1
    q.deQueue();

    q.enQueue(1);
    q.enQueue(2);
    q.enQueue(3);
    q.enQueue(4);
    q.enQueue(5);

    // Fails to enqueue because
    // front == 0 && rear == SIZE - 1
    q.enQueue(6);
    q.display();
    int elem = q.deQueue();
    if (elem != -1)
        cout << endl
             << "Deleted Element is " << elem;
    q.display();
    q.enQueue(7);
    q.display();

    // Fails to enqueue because front == rear + 1
    q.enQueue(8);
    return 0;
}
```

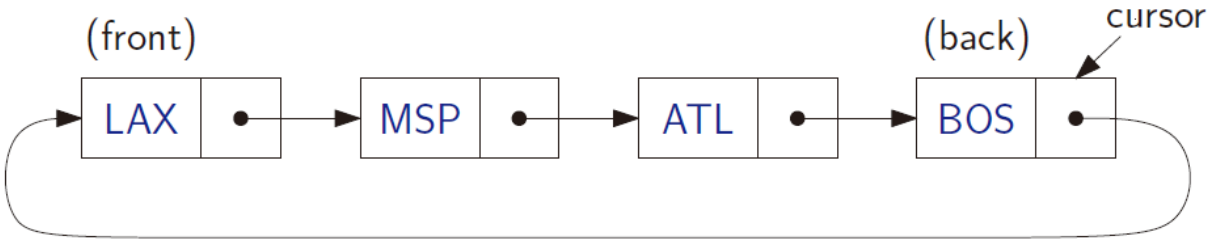
```
Queue is empty

Inserted 1

Inserted 2

Inserted 3

Inserted 4

Inserted 5
Queue is full
Front -> 0
Items -> 12345
Rear -> 4
```

```
Deleted Element is 1
Front -> 1
Items -> 2345
Rear -> 4

Inserted 7
Front -> 1
Items -> 23457
Rear -> 0
Queue is full
```

https://www.programiz.com/dsa/circular-queue

Circularly Linked List

- **Circularly Linked List**
  - Similar to a singly linked list (with the head and tail nodes)
  - The nodes are linked into a cycle
  - A special node called *cursor* is defined:
    - The back node is pointed by the cursor
    - The front node is pointed by the next node of the cursor



- **Supported Operations:**

These are correct (Textbook errors, Sec. 3.4.1)

back(): Return the element referenced by the cursor; an error results if the list is empty.

front(): Return the element immediately after the cursor; an error results if the list is empty.

advance(): Advance the cursor to the next node in the list.

add(e): Insert a new node with element e immediately after the cursor; if the list is empty, then this node becomes the cursor and its *next* pointer points to itself.

remove(): Remove the node immediately after the cursor (not the cursor itself, unless it is the only node); if the list becomes empty, the cursor is set to *null*.

Circularly Linked List

- **Circularly Linked List: C++ Implementation**

```
typedef string Elem;        // element type
class CNode {               // circularly linked list node
private:
  Elem elem;                // linked list element value
  CNode* next;              // next item in the list

  friend class CircleList;  // provide CircleList access
};
```

```
class CircleList {                          // a circularly linked list
public:
  CircleList();                             // constructor
  ~CircleList();                            // destructor
  bool empty() const;                       // is list empty?
  const Elem& front() const;                // element following cursor
  const Elem& back() const;                 // element at cursor
  void advance();                           // advance cursor
  void add(const Elem& e);                  // add after cursor
  void remove();                            // remove node after cursor
private:
  CNode* cursor;                            // the cursor
};
```

Circularly Linked List

- **Circularly Linked List: C++ Implementation**

```
CircleList::CircleList()                    // constructor
  : cursor(NULL) { }
CircleList::~CircleList()                   // destructor
  { while (!empty()) remove(); }


bool CircleList::empty() const              // is list empty?
  { return cursor == NULL; }
const Elem& CircleList::back() const        // element at cursor
  { return cursor->elem; }
const Elem& CircleList::front() const       // element following cursor
  { return cursor->next->elem; }
void CircleList::advance()                  // advance cursor
  { cursor = cursor->next; }
```

Circularly Linked List

- **Circularly Linked List: C++ Implementation**

```
void CircleList::add(const Elem& e) {        // add after cursor
  CNode* v = new CNode;                      // create a new node
  v->elem = e;
  if (cursor == NULL) {                      // list is empty?
    v->next = v;                             // v points to itself
    cursor = v;                              // cursor points to v
  }
  else {                                     // list is nonempty?
    v->next = cursor->next;                  // link in v after cursor
    cursor->next = v;
  }
}
```

```
void CircleList::remove() {                  // remove node after cursor
  CNode* old = cursor->next;                 // the node being removed
  if (old == cursor)                         // removing the only node?
    cursor = NULL;                           // list is now empty
  else
    cursor->next = old->next;                // link out the old node
  delete old;                                // delete the old node
}
```

Circularly Linked List

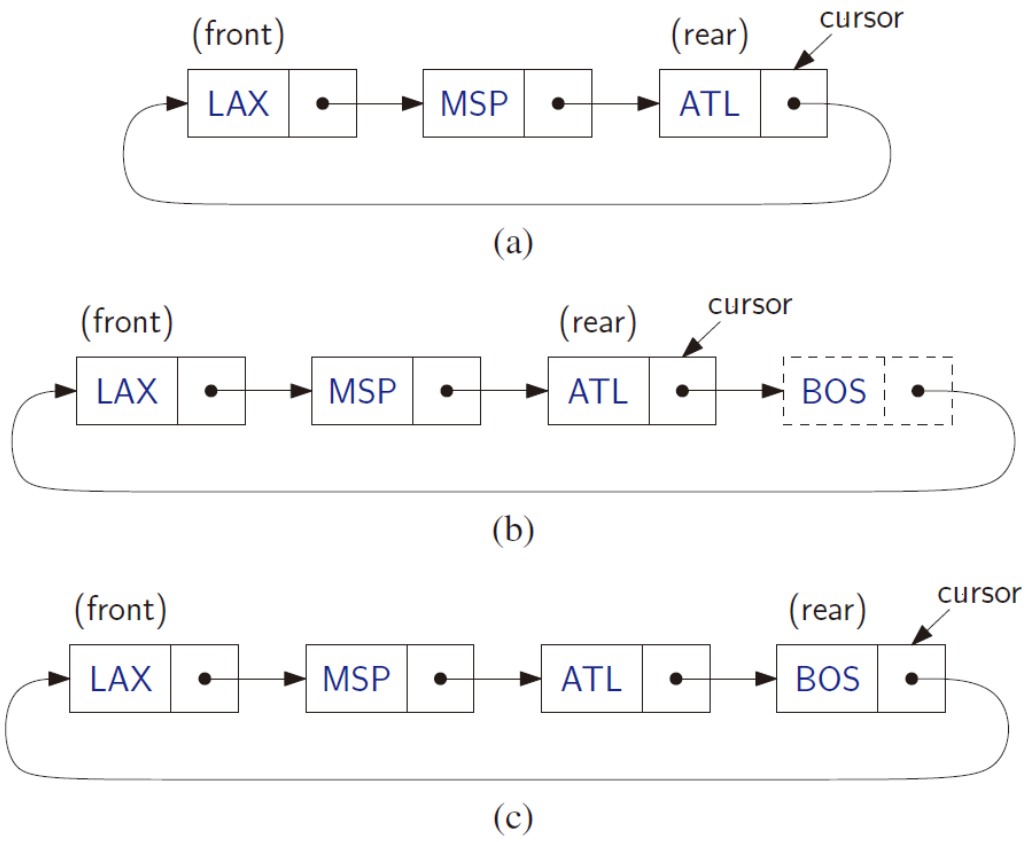
- **Circularly Linked List Example: Music Playlist**

```
int main() {
  CircleList playList;                  // []
  playList.add("Stayin Alive");         // [Stayin Alive*]
  playList.add("Le Freak");             // [Le Freak, Stayin Alive*]
  playList.add("Jive Talkin");          // [Jive Talkin, Le Freak, Stayin Alive*]

  playList.advance();                   // [Le Freak, Stayin Alive, Jive Talkin*]
  playList.advance();                   // [Stayin Alive, Jive Talkin, Le Freak*]
  playList.remove();                    // [Jive Talkin, Le Freak*]
  playList.add("Disco Inferno");        // [Disco Inferno, Jive Talkin, Le Freak*]
  return EXIT_SUCCESS;
}
```

Queue Implementation: Circularly Linked List

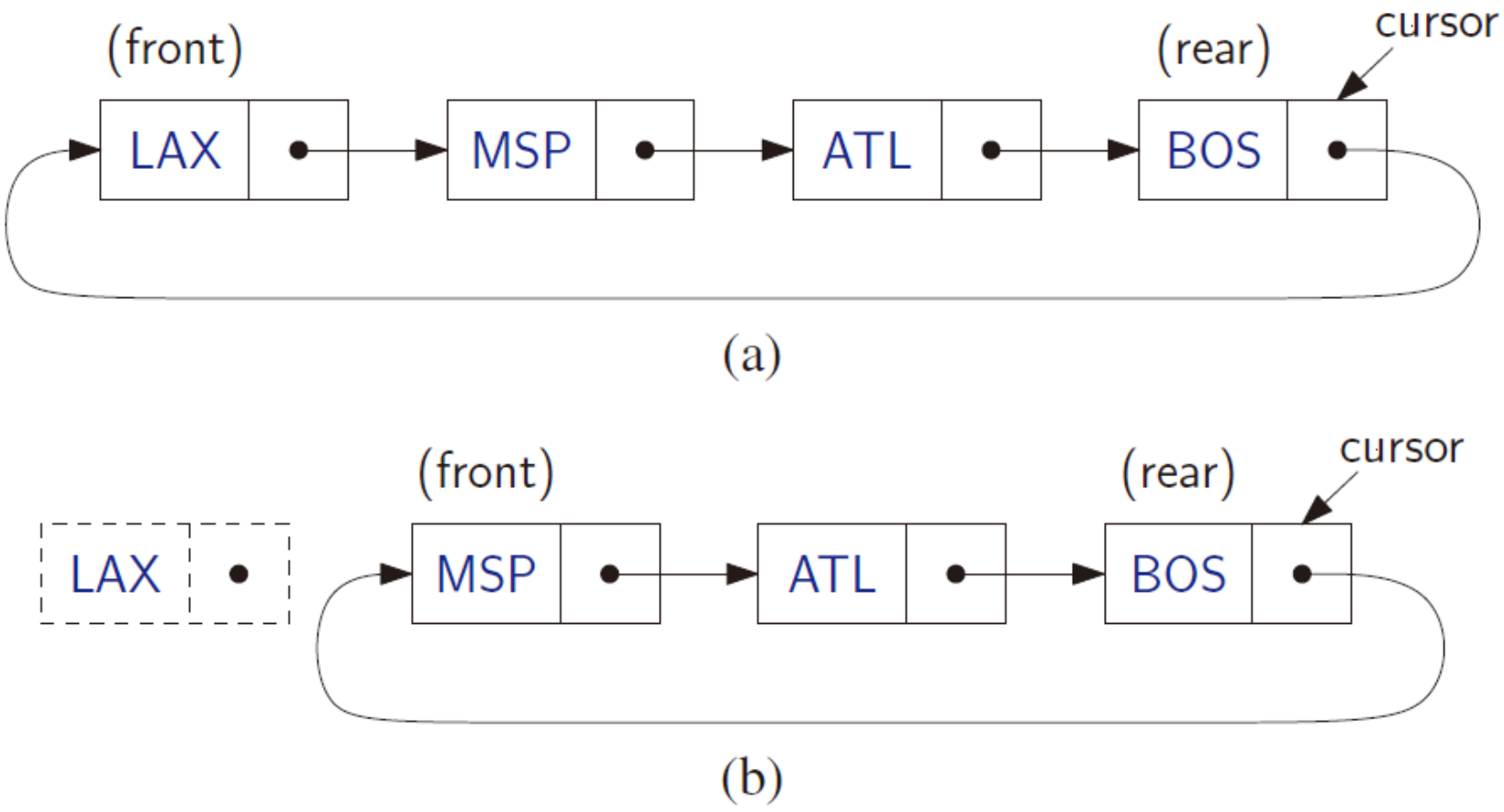
- **Correspondence**
  - The back node corresponds to the rear node
  - The front node corresponds to the front node

- **Enqueueing Operation**
  - Two function calls do the job:
    add() -> *advance()*



Queue Implementation: Circularly Linked List

- **Dequeueing Operation**
  - a single *remove()* call does the job



Queue Implementation: Circularly Linked List

- **C++ Implementation**

```
typedef string Elem;                                  // queue element type
class LinkedQueue {                                   // queue as circularly linked list
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:                                              // member data
  CircleList C;                                       // circular list of elements
  int n;                                              // number of elements
};
```

size() is not supported by the CircleList

Queue Implementation: Circularly Linked List

- **C++ Implementation**
  - Operations do not depends on the number of elements $O(1)$

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

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

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

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

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

                                              // dequeue element at front
void LinkedQueue::dequeue() throw(QueueEmpty) {
  if (empty())
    throw QueueEmpty("dequeue of empty queue");
  C.remove();                                 // remove from list front
  n--;
}
```