MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

19 Queues

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

March 7, 2022

Stack Example - Parentheses Matching Problem

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Stack Example - Parentheses Matching Problem

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Algorithm ParenMatch(X, n):

Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

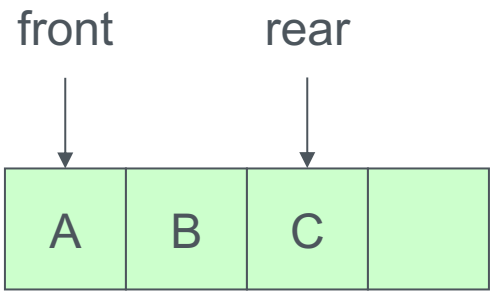
Output: **true** if and only if all the grouping symbols in X match

```
Let S be an empty stack
for i ← 0 to n − 1 do
    if X[i] is an opening grouping symbol then
        S.push(X[i])
    else if X[i] is a closing grouping symbol then
        if S.empty() then
            return false          {nothing to match with}
        if S.top() does not match the type of X[i] then
            return false          {wrong type}
        S.pop()
if S.empty() then
    return true                   {every symbol matched}
else
    return false                  {some symbols were never matched}
```

Queue

- A queue is a container of elements that are inserted and removed according to the First-In First-Out (FIFO) principle.
- Elements enter queue at the **rear** and *are removed* from the **front**

  "push" adds a new item on top of the stack.



Queue and its Applications

- Applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
  - As a data structure for algorithms to solve problems
    - BFS(Breadth First Search) for graphs
  - Component of other data structures

The Queue ADT

- The Queue ADT stores arbitrary objects
- Main stack 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.
  - size(): Return the number of elements in the queue
  - empty(): Return **true** if the queue is empty and **false** otherwise.

```
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
};
```

Exceptions for Queue ADT

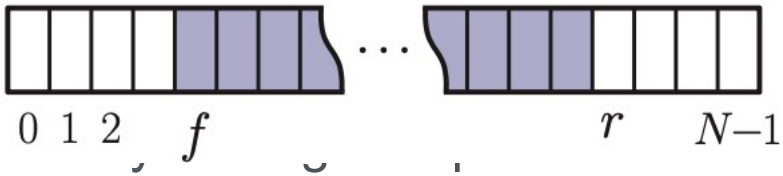
- In the Queue ADT, operations **dequeue** and **front** cannot be performed if the Queue is empty
- Attempting **dequeue** or **front** on an empty queue throws a **QueueEmpty** exception

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

```
class RuntimeException {                    // generic run-time exception
private:
  string errorMsg;
public:
  RuntimeException(const string& err) { errorMsg = err; }
  string getMessage() const { return errorMsg; }
};
```

Array-Based Queue

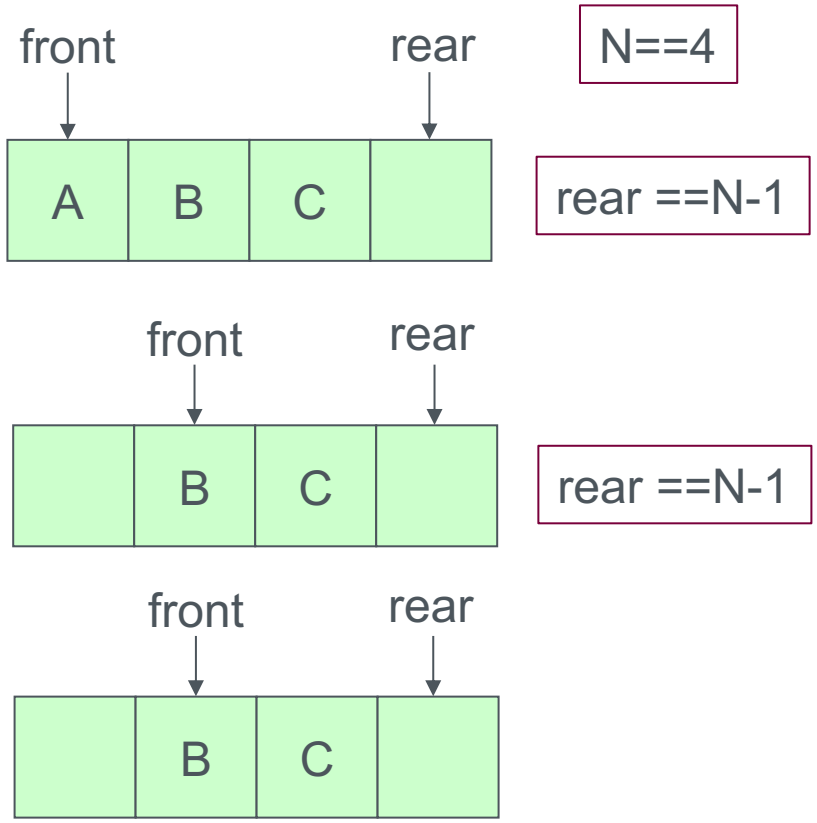
- An array with capacity N stores elements
- Two variables keeps track of the index of the front and rear element;
- **f**: index of front element
- **r**: index of element following rear element



- Th s may become **full**
- A **enqueue** operation will then throw a **QueueFull** exception

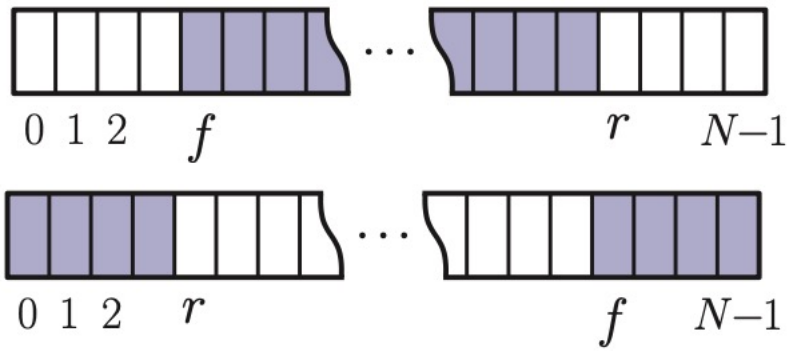
Array-Based Queue - An Issue

- **f**: index of front element
- **r**: index of element following rear element
- We can run into problem



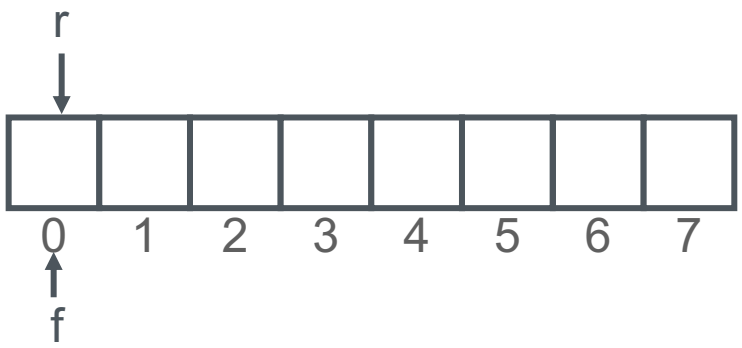
Array-Based Queue - Circular Array

- An array with capacity N stores elements
- **f**: index of front element
- **r**: index of element following rear element
- Each time we increment **f** or **r**, we simply need to compute this increment as:
  - (f + 1) mod N
  - (r + 1) mod N
- modulo operator in C++ is %



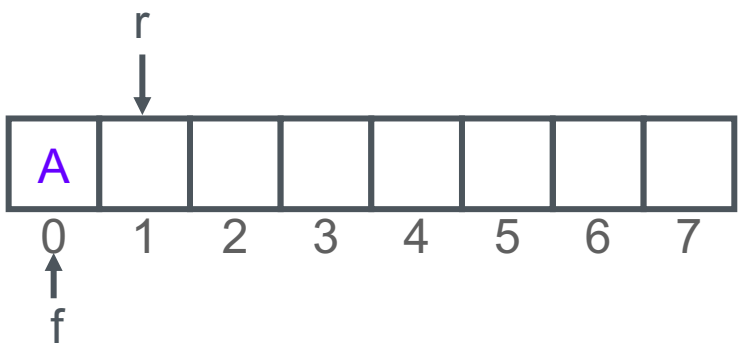
Algorithm size():
  return n
Algorithm empty():
  return $(n = 0)$
Algorithm front():
  if empty() **then**
    throw QueueEmpty exception
  return $Q[f]$
Algorithm dequeue():
  if empty() **then**
    throw QueueEmpty exception
  $f \leftarrow (f+1) \bmod N$
  $n = n - 1$
Algorithm enqueue(e):
  if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

Array-Based Queue - Circular Array



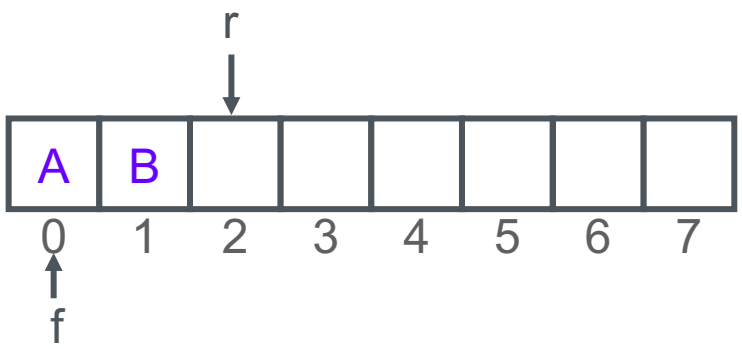
Enq (A)
Enq (B)
Enq (C)
Enq (D)
Deq ()
Deq ()
Deq ()
Enq (E)
Enq (F)
Enq (G)
Enq (H)
Enq (I)

Array-Based Queue - Circular Array



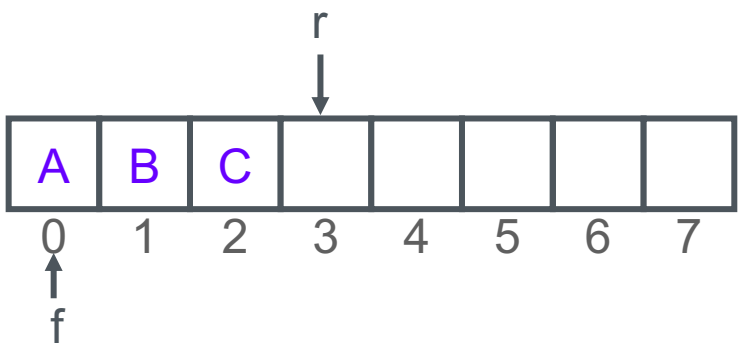
- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Circular Array



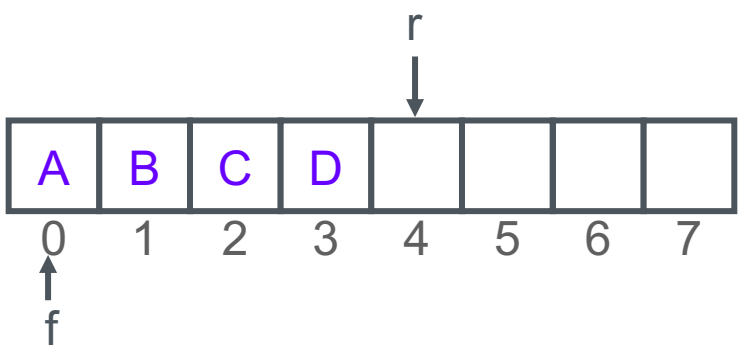
- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Circular Array



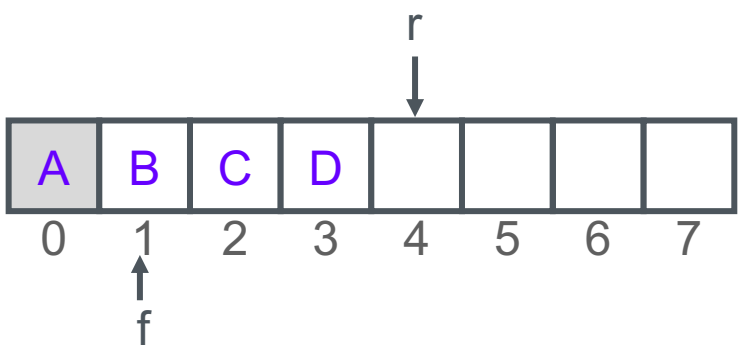
- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Circular Array



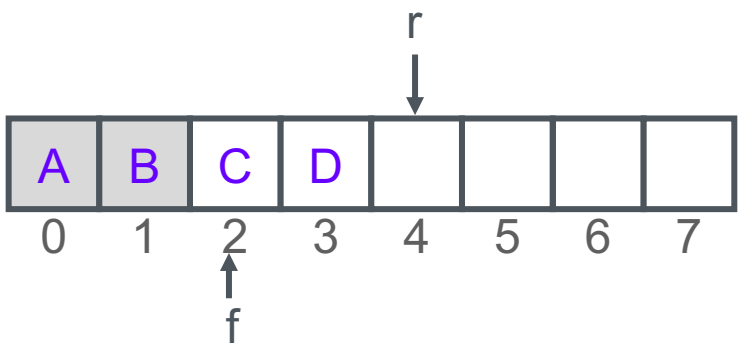
- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Circular Array



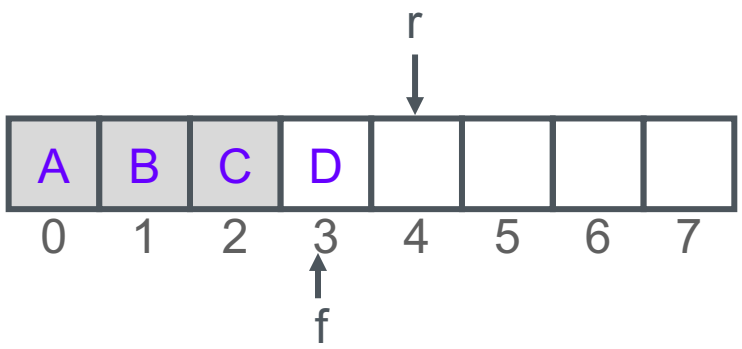
- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Circular Array



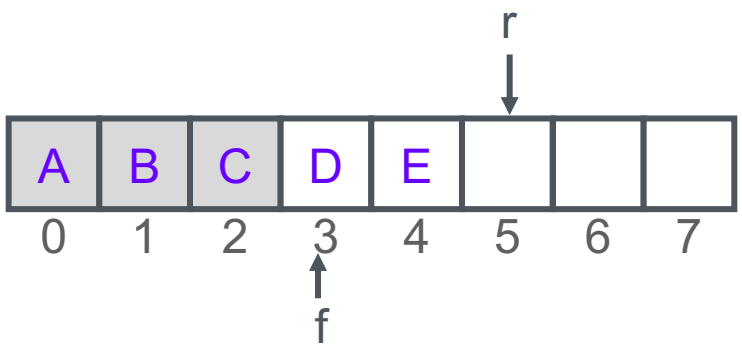
Enq (A)
Enq (B)
Enq (C)
Enq (D)
Deq ()
Deq ()
Deq ()
Enq (E)
Enq (F)
Enq (G)
Enq (H)
Enq (I)

Array-Based Queue - Circular Array



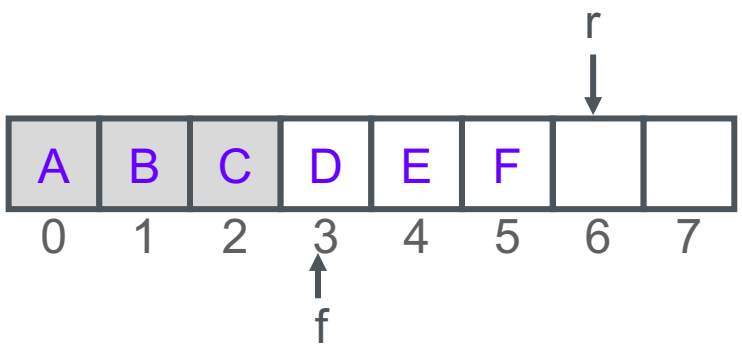
- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Circular Array



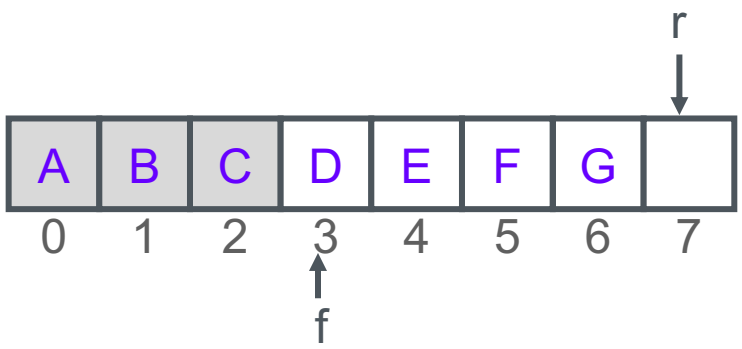
- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Circular Array



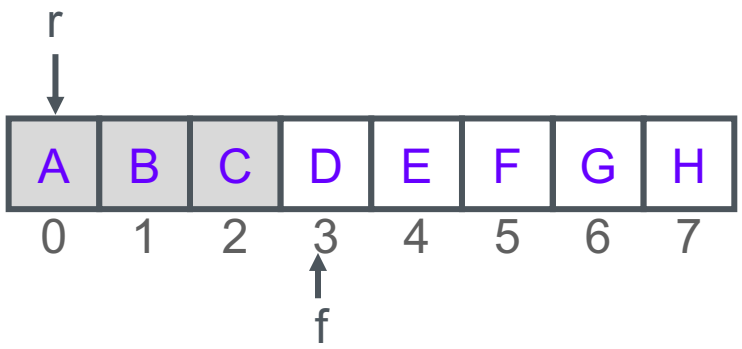
- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Circular Array



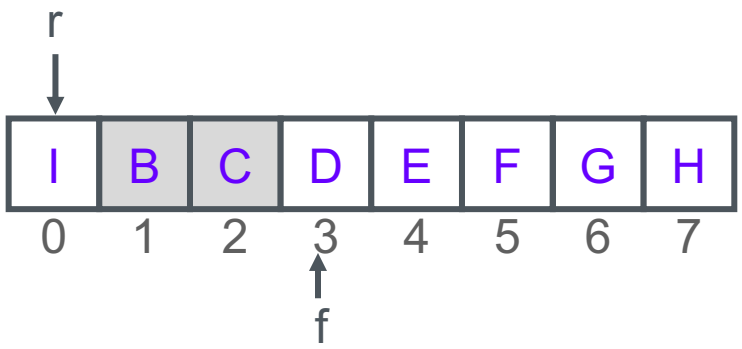
- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Circular Array



- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Circular Array



- Enq (A)
- Enq (B)
- Enq (C)
- Enq (D)
- Deq ()
- Deq ()
- Deq ()
- Enq (E)
- Enq (F)
- Enq (G)
- Enq (H)
- Enq (I)

Array-Based Queue - Performance

- Performance:
  - Let **n** be the number of elements in the Queue
  - The space used is **O(N)**
    - this is independent of the number of elements **n**
  - Each operation runs in time **O(1)**

- Limitations:
  - The maximum size of the Queue must be defined beforehand and cannot be changed
  - Trying to enqueue a new element into a full queue causes an implementation-specific exception (QueueFull Exception)

Circularly Linked List-Based Implementation

- We use that CircleList class
  - **C** stores the Queue elements
  - **n** stores the number of elements on Queue

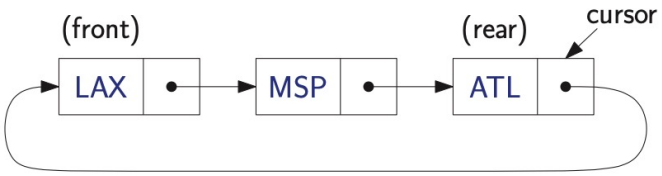
```
typedef string Elem;                                  // queue element type
class LinkedQueue {                                   // queue as doubly linked list
public:
  LinkedQueue();                                      // constructor
  int size() const;                                   // number 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
};
```

Circularly Linked List-Based Implementation

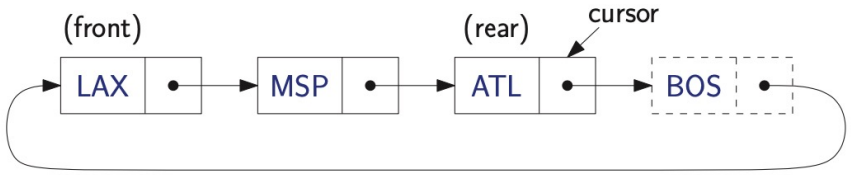
```
LinkedQueue::LinkedQueue()                    // constructor
  : C(), n(0) { }

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
}
```

Circularly Linked List-Based Implementation

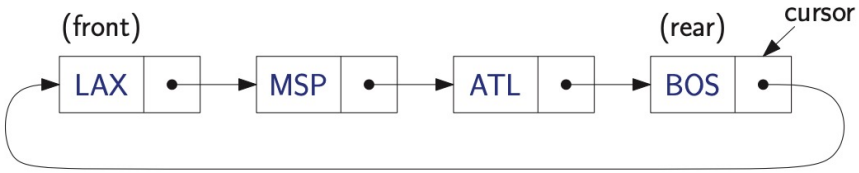
```
                                                    // enqueue element at rear
void LinkedQueue::enqueue(const Elem& e) {
  C.add(e);                                         // insert after cursor
  C.advance();                                      // ...and advance
  n++;
}
```



(a)



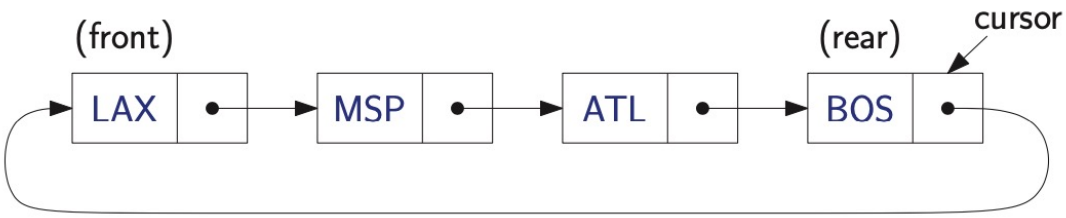
(b)



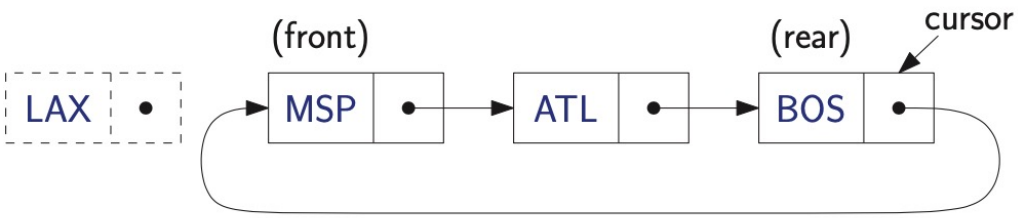
(c)

Circularly Linked List-Based Implementation

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



(a)



(b)

Questions?