DATA STRUCTURE & ALGORITHMS

Stack and Queue

STACK

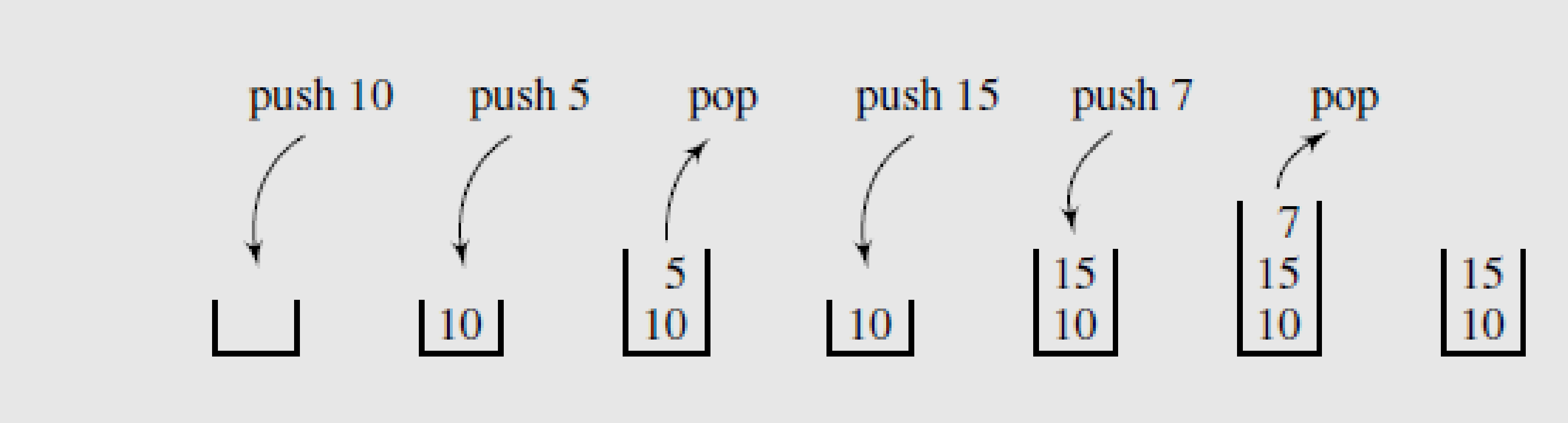
Content

- **Definition.**
- **Basic Stack Operations.**
- **Error Handling.**
- **Examples of Stacks.**
- **Parsing Arithmetic Expressions.**

Definition

- A ***stack*** is a data structure that consists of a sequence of data items of the same type.
- A stack is characterized by the property called ***last in, first out(LIFO)***.
  - That is, the last item put on the stack is the first item to be taken off.
  - All insertions and deletions of entries are made at one **end**, called the ***top*** of the stack.
  - A new item is placed in the stack by ***pushing*** it on the **top** of the stack.
  - Getting an item out of the stack is done by ***popping*** it out of from the top of the stack. Only the top item is accessible.

A series of operations executed on a stack.



Satck Operations

- A stack is defined in terms of operations that change its status and operations that check this status.
- The operations are as follows:
  - **clear()** **:** Clear the stack.
  - **isEmpty()** **:** Check to see if the stack is empty.
  - **push(el)** : Put the element el on the top of the stack.
  - **pop()** **:** Take the topmost element from the stack.
  - **topEl()** **:** Return the topmost element in the stack without removing it.

Satck Operations (Cont’d)

- **Initialize the Stack :**
  - Accepts a size, creates a new stack
  - Internally allocates an array of that many slots
  - Code :

```
class StackX
{
private int maxSize; // size of stack array
private long[] stackArray;
private int top; // top of stack
public StackX(int s) // constructor
{
maxSize = s; // set array size
stackArray = new long[maxSize]; // create array
top = -1; // no items yet}
```

Satck Operations (Cont'd)

- void **push(item)**
  - Increments top and stores a data item there :
  - Code:

```
public void push(long j) // put item on top of stack
{
stackArray[++top] = j; // increment top, insert item
}
```

- Item **pop()**
  - Returns the value at the top and decrements top
  - Note the value stays in the array! It's just inaccessible (why?)
  - Code:

```
public long pop() // take item from top of stack
{
return stackArray[top--]; // access item, decrement top
}
```

Satck Operations (Cont'd)

- void **PrintSatck()**
  - Print all element of sack in screen.
- void **peek()**
  - Return the value on top without changing the stack
- Boolean **isFull(),** Boolean **isEmpty()**
  - Return true or false

Error Handling

- What happens if you try to push an item onto a stack that's already full or pop an item from a stack that's empty?
- The responsibility for handling such errors up to the class user.
- The user should always check to be sure the stack is not full before inserting an item:

```
if( !theStack.isFull() )
insert(item);
else
System.out.print("Can't insert, stack is full");
```

Stack Examples

1. Word Reversal:

- Stacks can be used ***to reverse a sequence***.
  - For example, if a string ”**Computers**” is entered by the user the stack can be used to create and display the reverse string “**sretupmoC**” as follows.
  - The program simply ***pushes*** all of the characters of the string into the stack. Then it *pops* and ***display*** until the stack is empty.

Stack Examples **(Cont’d)**

2.Delimiter Matching:

- This is done in compilers!
- Parse text strings in a computer language
- Sample delimiters in C language :
  - {, }
  - [,]
  - (,)
- All opening delimiters should be matched by closing ones
- Also, later opening delimiters should be closer before earlier ones
  - See how the stack can help us here?

Example Strings

1. c[d]
2. a{b[c]d}e
3. a{b(c]d}e
4. a[b{c}d]e}
5. a{b(c)

- Which of these are correct?
- Which of these are incorrect?

3.Delimiter Matching Algorithm

- Read each character one at a time
- If an opening delimiter, place on the stack.
- If a closing delimiter, pop the stack
  - If the stack is empty, error
  - Otherwise if the opening delimiter matches, continue
  - Otherwise, error
- If the stack is not empty at the end, error

Example

- Let’s look at a stack for **a{b(c[d]e)f}**

Character	Stack	Action
a		x
{	{	push ‘{‘
b	{	x
(	{(	push ‘(‘
c	{(	x
[	{([	push ‘[‘
d	{([	x
]	{(	pop ‘[‘, match
e	{(	x
)	{	pop ‘(‘, match
f	{	x
}		pop ‘{‘, match

Parsing Arithmetic Expressions

- it’s fairly difficult, at least for a computer algorithm, to evaluate an arithmetic expression directly. It’s easier for the algorithm to use a two-step process:

1. Transform the arithmetic expression into a different format, called postfix notation.

2. Evaluate the postfix expression.

Postfix Notation

- Arithmetic expressions are written with an *operator* (+, –, *, or /) placed between two *operands* (numbers, or symbols that stand for numbers). This is called **infix notation**

- In **postfix notation** the operator *follows* the two operands.

- Example of Infix and Postfix and Prefix Expressions:

Infix	Postfix	Prefix
A+B	AB+	+AB
A+B-C	AB+C-	-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD

Translating Infix to Postfix

TABLE 4.3 Evaluating 3+4–5

Item Read	Expression Parsed So Far	Comments
3	3	
+	3+	
4	3+4	
–	7	When you see the –, you can evaluate 3+4.
	7–	
5	7–5	
End	2	When you reach the end of the expression, you can evaluate 7–5.

Translating Infix to Postfix

TABLE 4.4 Evaluating 3+4*5

Item Read	Expression Parsed So Far	Comments
3	3	
+	3+	
4	3+4	
*	3+4*	You can't evaluate 3+4 because * is higher precedence than +.
5	3+4*5	When you see the 5, you can evaluate 4*5.
	3+20	
End	23	When you see the end of the expression, you can evaluate 3+20.

Infix to Postfix: Algorithm

- Start with your infix expression, and an empty postfix string
  - Infix: 2*(3+4) Postfix:
- Go through the infix expression character-by-character
- For each operand:
  - Copy it to the postfix string
- For each operator:
  - Copy it at the 'right time'
  - When is this? We'll see

Example: 2*(3+4)

Read	***Postfix***	***Comment***
2	2	Operand
*	2	Operator
(	2	Operator
3	23	Operand
+		Operator
4	234	Operand
)	234+	Saw), copy +
	234+*	Copy remaining ops

Example: 3+4*5

Read	*Postfix*	*Comment*
3	3	Operand
+	3	Operator
4	34	Operand
*	34	Operator
5	345	Operand
	345*	Saw 5, copy *
	345*+	Copy remaining ops

Rules on copying operators

- You cannot copy an operator to the postfix string if:
  - It is followed by a left parenthesis ‘(‘
  - It is followed by an operator with higher precedence (i.e., a ‘+’ followed by a ‘*’)
- If neither of these are true, you can copy an operator once you have copied both its operands
- We can use a stack to hold the operators before they are copied. Here’s how:

How can we use a stack?

- Suppose we have our infix expression, empty postfix string and empty stack S. We can have the following rules:
  - If we get an operand, copy it to the postfix string
  - If we get a ‘(‘, push it onto S
  - If we get a ‘)’:
    - Keep popping S and copying operators to the postfix string until either S is empty or the item popped is a ‘(‘
  - Any other operator:
    - If S is empty, push it onto S
    - Otherwise, while S is not empty and the top of S is not a ‘(‘ or an operator of lower precedence, pop S and copy to the postfix string
    - Push operator onto S

  - To convince ourselves, let’s try some of the expressions

QUEUE

DEFINITION OF QUEUE

- A **Queue** is a data structure that consists sequence of data items of the same type.

- A **Queue** is characterized by the property called ***first in, first out(FIFO)***.
  - The first element inserted into the queue is the first element to be removed.
  - An items may be deleted at one end (called the ***front*** of the queue) .
  - An items may be inserted at the other end (the ***rear*** of the queue).

QUEUE

items[MAXQUEUE-1]		
.	.	
.	.	
.	.	
items[2]	C	← **Rear=2**
items[1]	B	
items[0]	A	← **Front=0**

Applications OF QUEUE

- Graph searching
- Simulating real-world situations
  - People waiting in bank lines.
  - Airplanes waiting to take off.
  - Packets waiting to be transmitted over the internet.
- Hardware
  - Printer queue.
  - Keyboard strokes.

Implementation of the QUEUE

- The queue can be implemented by the use of ***arrays*** and ***linked lists***.

BASIC QUEUE OPERATIONS

- Operations include
- ***clear()***—Clear the queue.
- ***isEmpty()***—Check to see if the queue is empty.
- ***enqueue(el)***—Put the element *el* at the end of the queue.
- ***dequeue()***—Take the first element from the queue.
- ***firstEl()***—Return the first element in the queue without removing it.

BASIC QUEUE OPERATIONS

- **Definition and Initialize the queue**
  - The queue is initialized by having the ***rear*** set to ***-1***, and ***front*** set to ***0***.

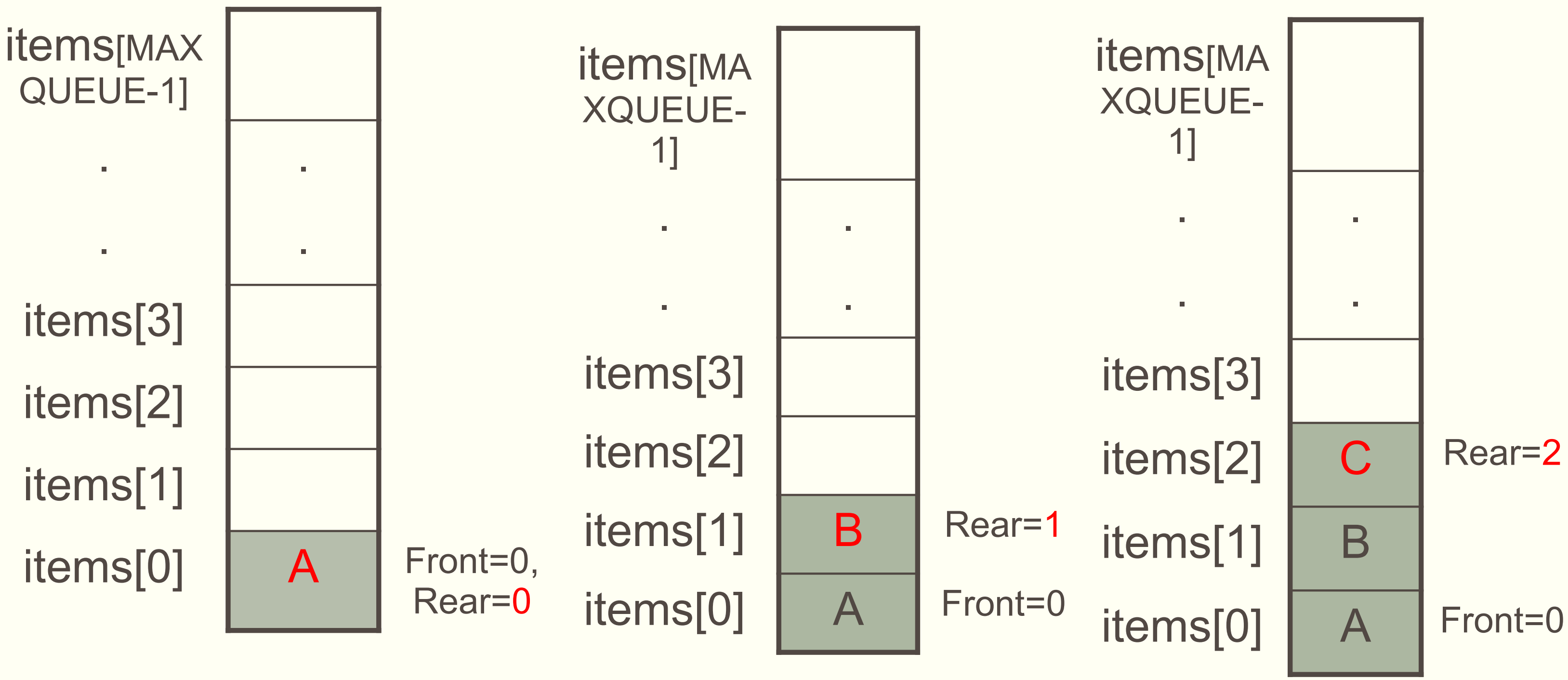
```
class Queue
{private int maxSize;
private long[] queArray;
private int front;
private int rear;
private int nItems;
public Queue(int s) // constructor
{maxSize = s;
queArray = new long[maxSize];
front = 0;
rear = -1;
nItems = 0;}
```

QUEUE OPERATIONS

- The two basic queue operations are:
  - **inserting (put or add or enque)** an item, which is placed at the rear of the queue, and
  - ***Removing (delete*** **or** ***get*** **or** ***deque)*** an item, which is taken from the front of the queue.

Insert to the queue

- **an item (*A*) is *inserted* at the *Rear* of the queue.**
- **an item (*B*) is *inserted* at the *Rear* of the queue.**
- **an item (*C*) is *inserted* at the *Rear* of the queue**



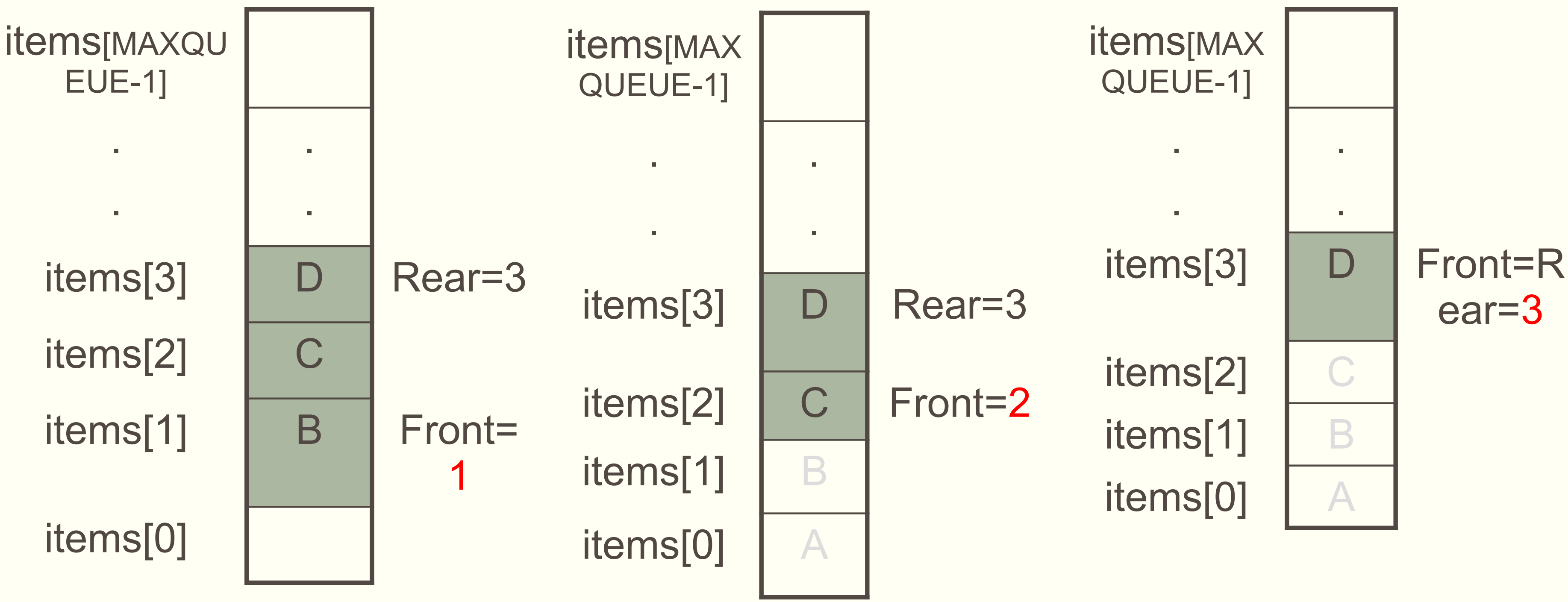
Insert to the queue **(Cont'd)**

- Void **insert (**item)
  - Also referred to as put(), add(), or enque()
  - Inserts an element at the back of the queue
  - Code:

```
public void insert(long j) // put item at rear of queue
{
if(! rear == maxSize-1) // deal with wraparound
rear = -1;
queArray[++rear] = j; // increment rear and insert
nItems++; // one more item
}
```

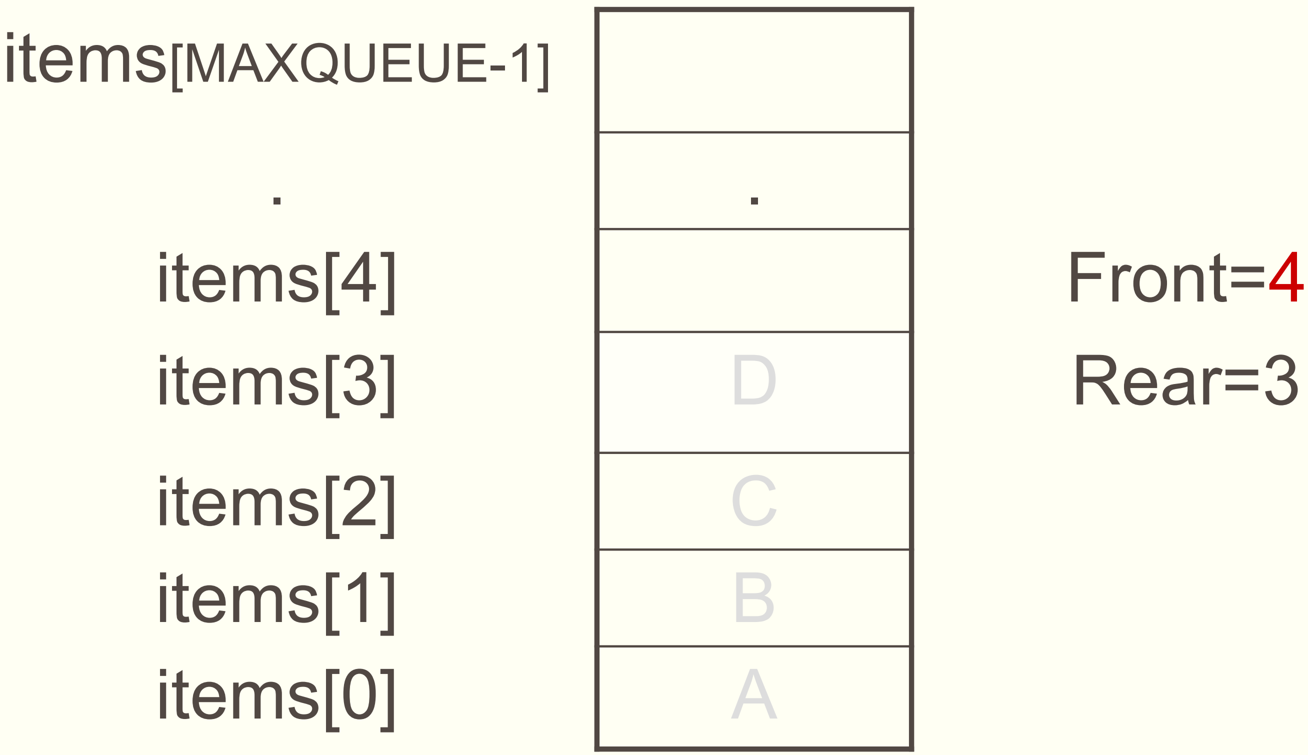
Remove from the queue

- **an item (*A*) is removed (deleted) from the *Front* of the queue.**
- ***Remove* one more item from the front of the queue.**
- ***Remove* one more item from the front of the queue.**



Remove from the queue (Cont'd)

- ***Remove* one more item from the front of the queue.**



Remove from the queue (Cont'd)

- Item **remove** (void)
  - Also referred to as get(), delete(), or deque()
  - Removes an element from the front of the queue.
  - Code:

```
public long remove() // take item from front of queue
{
long temp = queArray[front++]; // get value and incr front
nItems--; // one less item
return  temp;
}
```

Queue Operations (Cont'd)

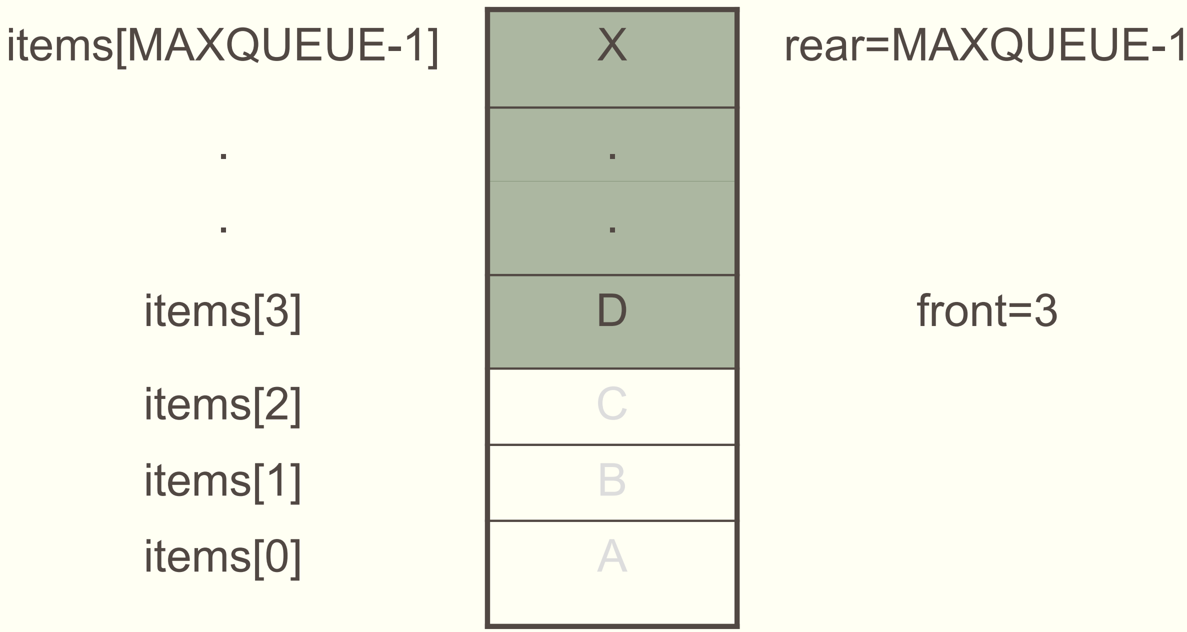
- void **PrintQueue()**
  - Print all element of Queue in screen.
- Item **peekRear** (void)
  - Element at the back of the queue
- Item **peekFront** (void)
  - Element at the front of the queue.
- Boolean **isFull(),** Boolean **isEmpty()**
  - Return true or false

ERROR HANDLING

- In terms of memory now, what about the queue do we need to worry about?
  - That we did not have to worry about with the stack
  - Hint: Think
  - in terms of the low-level representation

INSERT / REMOVE ITEMS

- ***Assume that* the *rear*= MAXQUEUE-1**

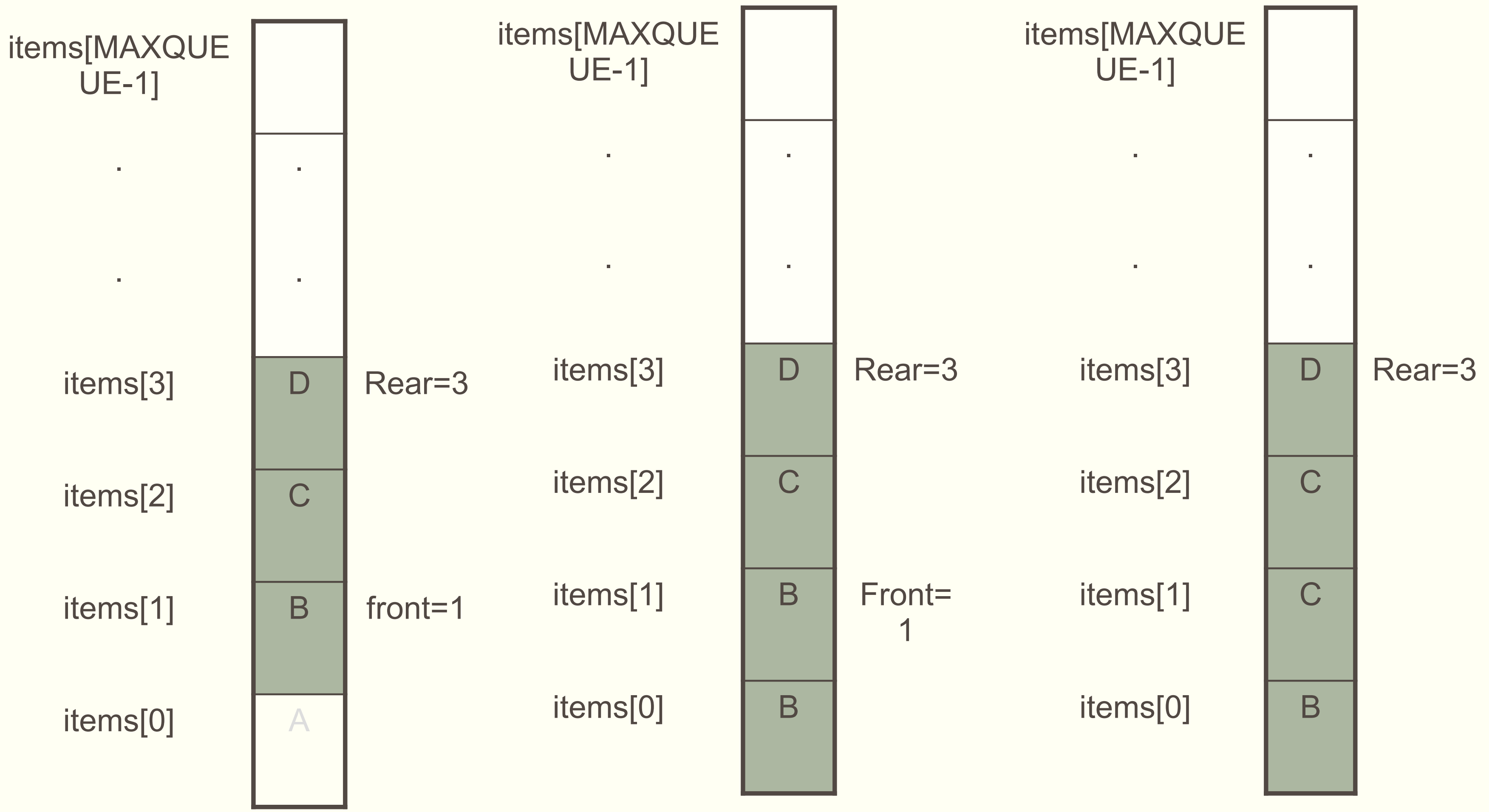


- **What happens if we want to insert a new item into the queue?**

INSERT / REMOVE ITEMS

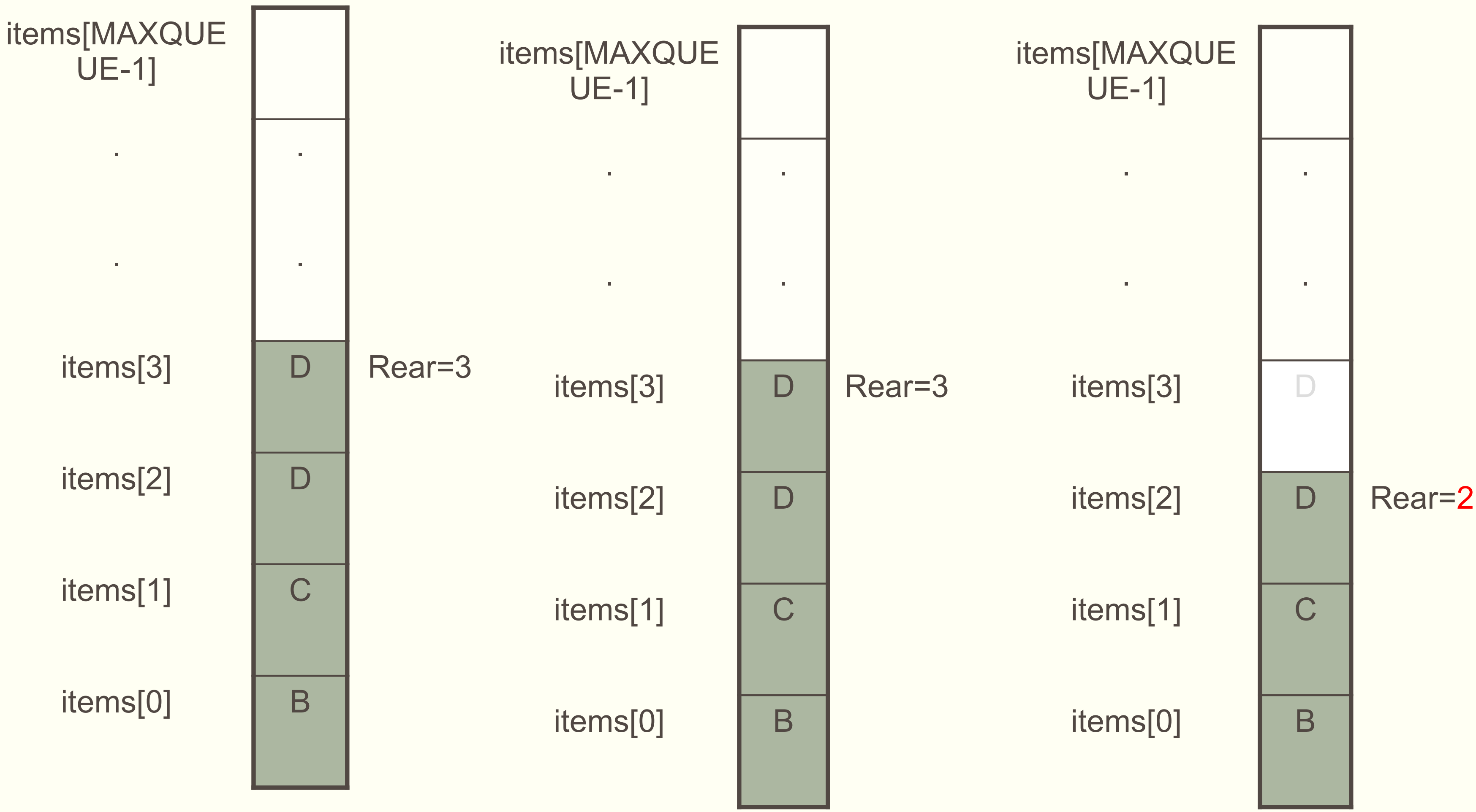
- What happens if we want to insert a new item F into the queue?
- Although there is some empty space, the queue is full.
- One of the methods to overcome this problem is to shift all the items to occupy the location of deleted item.

REMOVE ITEM



items[MAXQUEUE-1]
.
.
items[3]
items[2]
items[1]
items[0]
D
C
B
A
Rear=3
front=1
items[MAXQUEUE-1]
.
.
items[3]
items[2]
items[1]
items[0]
D
C
B
B
Rear=3
Front=1
items[MAXQUEUE-1]
.
.
items[3]
items[2]
items[1]
items[0]
D
C
C
B
Rear=3

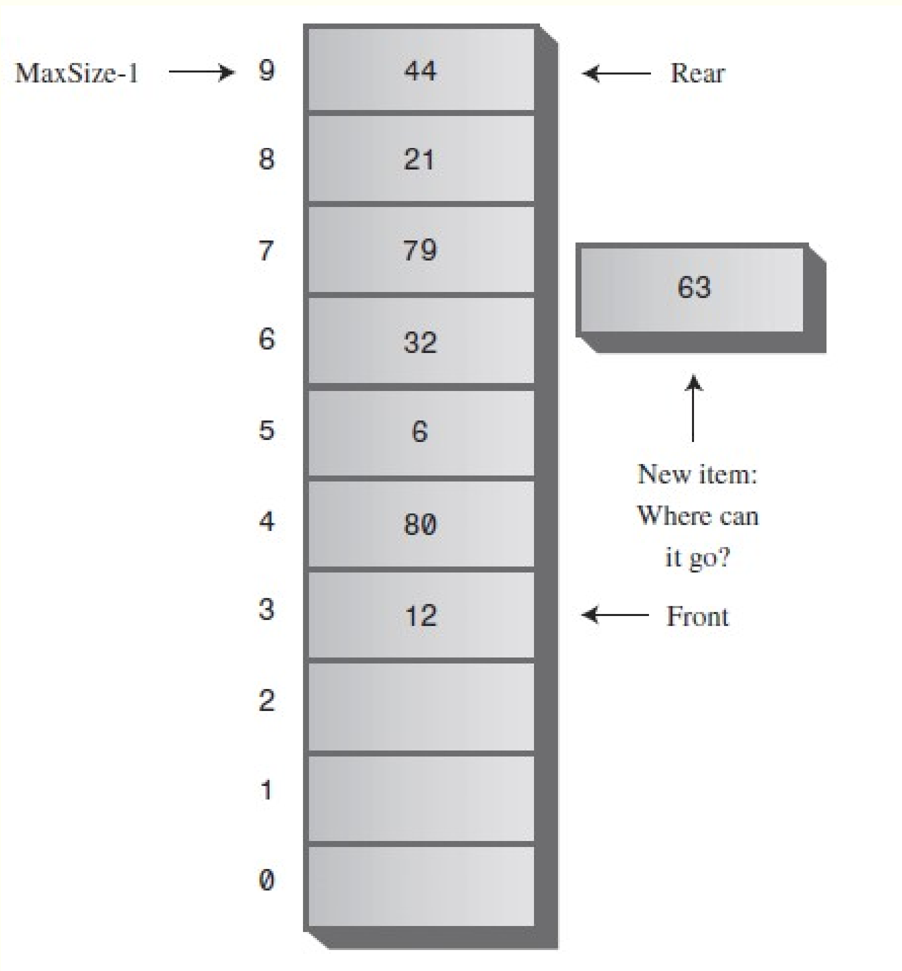
REMOVE ITEM

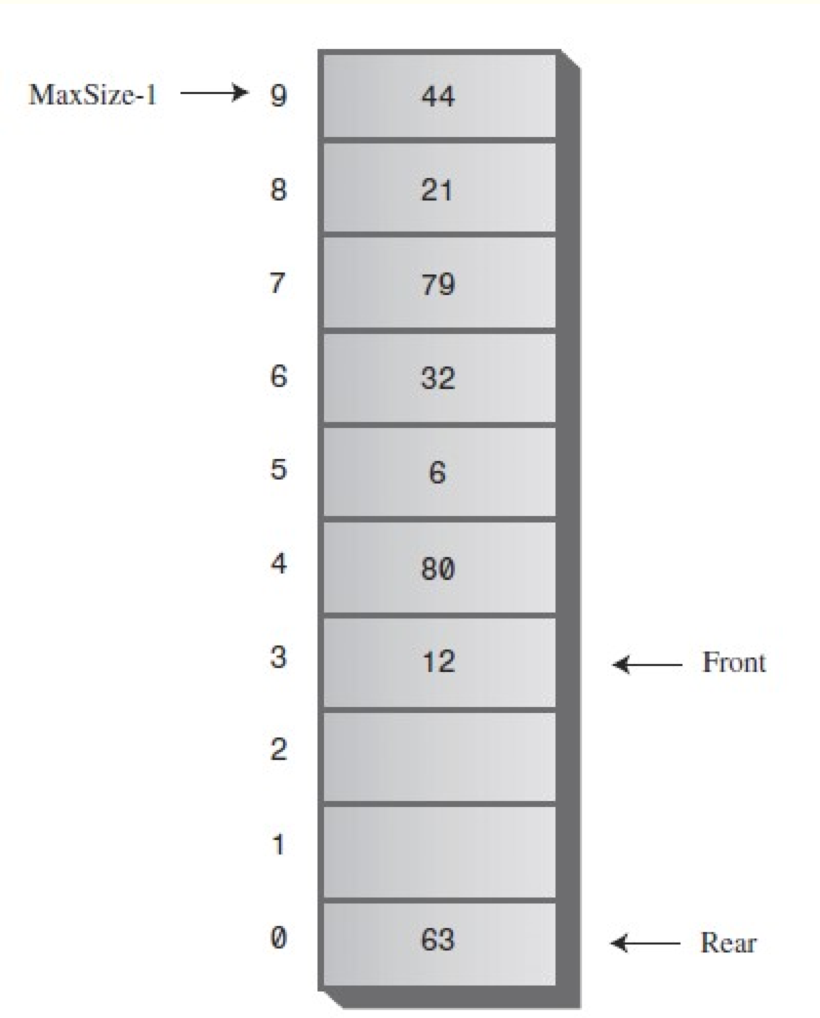


items[MAXQUEUE-1]
.
.
items[3]
items[2]
items[1]
items[0]
D
D
C
B
Rear=3
items[MAXQUEUE-1]
.
.
items[3]
items[2]
items[1]
items[0]
D
D
C
B
Rear=3
items[MAXQUEUE-1]
.
.
items[3]
items[2]
items[1]
items[0]
D
D
C
B
Rear=2

INSERT / REMOVE ITEMS

- Since all the items in the queue are required to shift when an item is deleted, this method is not preferred.
- The other method is ***circular queue.***
- When rear = MAXQUEUE-1, the next element is entered at items[0] in case that spot is free.


MaxSize-1
9
8
7
6
5
4
3
2
1
0
44
21
79
32
6
80
12
Rear
63
New item:
Where can
it go?
Front


MaxSize-1
9
8
7
6
5
4
3
2
1
0
44
21
79
32
6
80
12
63
Front
Rear