CHAPTER 2: STACKS AND QUEUES

BIBHA STHAPIT
ASST. PROFESSOR
IoE, PULCHOWK CAMPUS

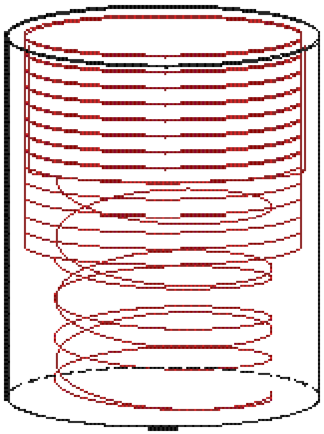
Stack : Introduction

- A ***stack*** is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end called the ***top*** of the stack.

- The first inserted element can be removed only at last and the last inserted element first. This condition of operation is known as ***Last-In-First-Out (LIFO)***.

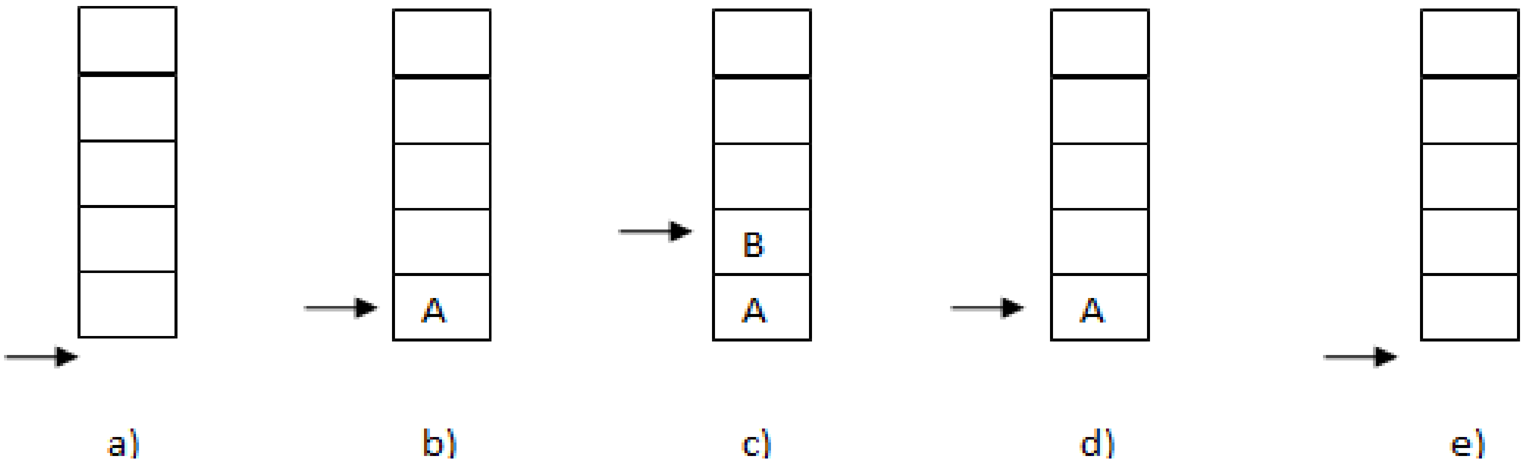
- Stacks are linear data structures and

hold objects, usually all of the same type.



Stack : Introduction

Example:



a) Stack is empty

b) Insert item A in stack

c) Insert item B in stack

d) Remove item B from stack

e) Remove Item A from stack

→ Top pointer

Operations on Stack

- Push
  - Adding an item
- Pop
  - Removing an item
- Peek
  - Returns top element of the stack
- Display
  - Displays all elements of the stack

- **Overflow and Underflow conditions:**
- When we try to insert an item in a full stack, stack is ***overflow*** and the result of an illegal attempt to pop an item from an empty stack is known as ***underflow***.

Push Operation

- A new item (*A*) is *inserted* at the *Top* of the stack

items[MAX-1]

.

.

items[3]

items[2]

items[1]

items[0] A ← Top=0

- A new item (*B*) is *inserted* at the *Top* of the stack

items[MAX-1]

. .

. .

items[3]

items[2]

items[1] B ← Top=1

items[0] A

Push Operation

- A new item (*C*) is *inserted* at the *Top* of the stack

items[MAX-1] .

. .

.

items[3]

items[2] C ← Top=2

items[1] B

items[0] A

- A new item (*D*) is *inserted* at the *Top* of the stack

items[MAX-1]

. . .

. . .

items[3] D ← Top=3

items[2] C

items[1] B

items[0] A

Push Operation

- Array implementation
- Initial condition : TOP= -1

PUSH(STACK, N, TOP, ITEM)

1. If TOP=N-**1**:

   write OVERFLOW, and Return.

2. Set TOP := TOP + 1.
3. Set STACK[TOP]:= ITEM.
4. Return.

Pop Operation

- an item (*D*) is deleted from the *Top* of the stack

items[MAX-1]		
.	.	
.	.	
items[3]	D	
items[2]	C	← Top=2
items[1]	B	
items[0]	A	

- an item (*C*) is deleted from the *Top* of the stack

items[MAX-1]		
.	.	
.	.	
items[3]	D	
items[2]	C	
items[1]	B	← Top=1
items[0]	A	

Pop Operation

- a n item (*B*) is deleted from the *Top* of the stack

items[MAX-1]

. .

. .

items[3] D

items[2] C

items[1] B

items[0] A ⟵ Top=0

- a n item (*A*) is deleted from the *Top* of the stack

items[MAX-1]

. .

. .

items[3] D

items[2] C

items[1] B

items[0] A

⟵ Top=-1

Pop Operation

- Array Implementation

POP(STACK, N, TOP, ITEM)

1. If TOP = **-1** then :

   write: UNDERFLOW, and Return.

2. Set ITEM := STACK [TOP].
3. Set TOP := TOP - **1**.
4. Return.

```
void StackType::Push(ItemType newItem)
{
if (IsFull()) throw FullStack();
top++;
items[top] =newItem;
 }

ItemType StackType::Pop()
{
if(IsEmpty()) throw EmptyStack();
return items[top];
top--;
}
```

Prefix and postfix notations

- Polish notation, also known as prefix notation.
- RPN=Reverse Polish Notation=Postfix notation

- It is a symbolic logic invented by **Polish** mathematician **Jan Lukasiewicz** in the 1920's.

- Most compilers convert an expression in infix notation to ***postfix*** where the operators are written <u>after</u> the operands

- So a * b + c becomes ab * c +
- Advantage: expressions can be written without parentheses

Prefix and postfix examples

INFIX	POSTFIX	PREFIX
A + B	A B +	+ A B
A * B + C	A B * C +	+ * A B C
A * (B + C)	A B C + *	* A + B C
A - (B - (C - D))	A B C D---	-A-B-C D
A - B - C - D	A B-C-D-	---A B C D

Prefix : Operators come before the operands

Transforming infix to postfix

- *By hand:* "Fully parenthesize-move-erase" method:

1. Fully parenthesize the expression.
2. Replace each right parenthesis by the corresponding operator.
3. Erase all left parentheses.

Examples:

`A * B + C` → `((A * B) + C)`
→ `((A B * C +`
→ `A B * C +`

`A * (B + C)` → `(A * (B + C) )`
→ `(A (B C + *`
→ `A B C + *`

Transforming infix to prefix

- *By hand:* "Fully parenthesize-move-erase" method:

1. Fully parenthesize the expression.
2. Replace each left parenthesis by the corresponding operator.
3. Erase all right parentheses.

Examples:

```
A * B + C  → ((A * B) + C)
           → + * A B) C)
           → + * A B C
```

```
A * (B + C)  → (A * (B + C))
             → * A + B C ))
             → * A + B C
```

Infix to postfix using stack

- 1. Scan the infix expression from left to right
- 2. If the scanned character is operand, then it will be added to postfix expression
- 3. If the scanned character is either operator or left parenthesis, then it will be pushed to stack
  - 3.a. If the priority of scanned operator is greater than that of stack operator, then the scanned operator is also pushed to stack
  - 3.b. If the priority of scanned operator is less than or equal to that of stack operator, then the stacked operator is popped from stack and added to expression while scanned operator is pushed to stack
- If the scanned character is right parenthesis, then operators from stack are popped to postfix expression until left parenthesis is not encountered.
- NOTE: use parenthesis for matching only, do not add it to expression

Infix to postfix using stack: Algorithm

- **POLISH (Q, P)**

1. PUSH "(" on to STACK and add ")" to the end of Q.
2. Scan Q from left to right and Repeat steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator is encountered, then:

(a) Repeatedly POP from STACK and add to P each operator (On the TOP of STACK) which has the same precedence as or higher precedence than @.

(b) Add @ to STACK.

[End of If structure.]

6. If a right parenthesis is encountered, then:

   (a) Repeatedly POP from STACK and add to P each operator (On the TOP of STACK.) until a left parenthesis is encountered.

   (b) Remove the left parenthesis. [Don't add the left parenthesis to P.] [End of If Structure.]

   - [End of step 2 Loop.]

7. Exit.

Infix Character Scanned	STACK	Postfix Expression
	(	
A	(	A
-	(-	A
(	(- (	A
B	(- (	A B
/	(- (/	A B
C	(- (/	A B C
+	(- (+	A B C /
(	(- (+ (	A B C /
D	(- (+ (	A B C / D
%	(- (+ (%	A B C / D
E	(- (+ (%	A B C / D E
*	(- (+ (% *	A B C / D E
F	(- (+ (% *	A B C / D E F
)	(- (+	A B C / D E F * %
/	(- (+ /	A B C / D E F * %
G	(- (+ /	A B C / D E F * % G
)	(-	A B C / D E F * % G / +
*	(- *	A B C / D E F * % G / +
H	(- *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * -

Chapter 2

Evaluating RPN expressions

- *"By hand" (Underlining technique)*:

1. Scan the expression from left to right to find an operator.
2. Locate ("underline") the last two preceding operands and combine them using this operator.
3. Repeat until the end of the expression is reached.

- Example:

```
• 2 3 4 + 5 6 - - *
→   2 3 4 + 5 6 - - *
 → 2 7 5 6 - - *
 → 2 7 5 6 - - *
 → 2 7 -1 - *
 → 2 7 -1 - *   → 2 8 * → 2 8 * → 16
```

Evaluating RPN expressions

- P is an arithmetic expression in Postfix Notation.

1. Add a right parenthesis ")" at the end of P.
2. Scan P from left to right and Repeat Step 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator @ is encountered, then:
   (a) Remove the two top elements of STACK, where A
   - is the top element and B is the next to top element.

   (b) Evaluate B @ A.
   (c) Place the result of (b) back on STACK.
   [End of if structure.]

- [End of step 2 Loop.]

5. Set VALUE equal to the top element on STACK.
6. Exit.

Expression	*Stack*	*Comments*
2 4 * 9 5 + −	2 ← top	Push 2 onto the stack.
4 * 9 5 + −	4 ← top, 2	Push 4 onto the stack.
* 9 5 + −	8 ← top	Pop 4 and 2 from the stack, multiply, and push the result back onto the stack.
9 5 + −	9 ← top, 8	Push 9 onto the stack.
5 + −	5 ← top, 9, 8	Push 5 onto the stack.
+ −	14 ← top, 8	Pop 5 and 9 from the stack, add, and push the result back onto the stack.
−	−6 ← top	Pop 14 and 8 from the stack, subtract, and push the result back onto the stack.
(end of string)	−6 ← top	Value of expression is on top of the stack.

Evaluate postfix expression:
5,6,2,+,*,12,4,/,-

	Symbol Scanned	Stack	Operation (B op A)
(1)	5	5	
(2)	6	5, 6	
(3)	2	5, 6, 2	
(4)	+	5, 8	[6+2] (A=2, B=6)
(5)	*	40	[5*8] (A=8, B=5)
(6)	12	40, 12	
(7)	4	40, 12, 4	
(8)	/	40, 3	[12/4] (A=4, B=12)
(9)	-	37	[40-3] (A=3, B=40)

Matching the nested parentheses

- Checking the validity od arithmetic expressions by matching parentheses.

- Parentheses are nested correctly if:
  - There equal number of right and left parentheses.
  - Every right parentheses is preceded by a matching left parentheses.

- *Example:*
- (a+b*(c+d)-e *invalid*
- a+((c+d)-e*f)) *invalid*
-)a+b(*invalid*
- (a+b))-c+d *invalid*

Matching the nested parentheses : Algorithm

- Input the arithmetic expression.
- Scan the expression left to right character-by-character.
- During your scanning,
  - if you found a left parenthesis, push it onto the stack and continue scanning
  - if you found a right parenthesis, examine the status of the stack,
    - if the stack is empty, then the right parenthesis does not have a matching left parenthesis.
    - if the stack is non-empty, pop the stack item and continue scanning.
- if expression end, examine the status of the stack,
  - if the stack is empty, then the expression is correct.
  - otherwise one or more left parentheses have been opened and have not been closed

Queue : Introduction

- A **Queue** is an ordered collection of items from which items may be deleted at one end (called the ***front*** of the queue) and into which items may be inserted at the other end (the ***rear*** of the queue).

- The first element inserted into the queue is the first element to be removed. For this reason a queue is sometimes called a **FIFO** (first-in first-out) list.

- Enqueue: Inserting an item
- Dequeue: Deleleting an item

Enqueue example

- A new item (*A*) is *inserted* at the *Rear* of the queue
- A new item (*B*) is *inserted* at the *Rear* of the queue

items[MAXQUEUE -1]			items[MAXQUEUE -1]		
.			.		
.			.		
items[3]			items[3]		
items[2]			items[2]		
items[1]			items[1]	B ←	Rear=1
items[0]	A ←	Front=0, Rear=0	items[0]	A ←	Front=0

Enqueue example

- A new item (*C*) is *inserted* at the *Rear* of the queue

items[MAXQUEUE -1]

.

.

items[3]

items[2] C Rear=2

items[1] B

items[0] A Front=0

- A new item (*D*) is *inserted* at the *Rear* of the queue

items[MAXQUEUE -1]

. .

. .

items[3] D Rear=3

items[2] C

items[1] B

items[0] A Front=0

Enqueue operation

- QINSERT(QUEUE, N, FRONT, REAR, ITEM)

1. If REAR = N-1,

   then : write OVERFLOW, and Return.

2. If FRONT = REAR = NULL, then:
   - Set FRONT := 0and REAR := 0
   - Else :
   - Set REAR := REAR + 1.
3. Set QUEUE[REAR]:= ITEM.
4. Return.

Dequeue example

- An item (*A*) is deleted from the *Front* of the queue

items[MAXQUEUE
-1]

.   .

.   .

items[3]	D	Rear=3
items[2]	C	
items[1]	B	Front=1
items[0]	A	

- An item (*B*) is deleted from the *Front* of the queue

items[MAXQUEUE
-1]

.   .

.   .

.

items[3]	D	Rear=3
items[2]	C	Front=2
items[1]	B	
items[0]	A	

Dequeue example

- An item (*C*) is deleted from the *Front* of the queue

items[MAXQUEUE
-1]

. .

. .

items[3] D Rear=3
Front=3

items[2] C

items[1] B

items[0] A

- An item (*D*) is deleted from the *Front* of the queue

items[MAXQUEUE
-1]

. .

. .

.

Front=4

items[3] D Rear=3

items[2] C

items[1] B

items[0] A

Dequeue Operation

- QDELETE(QUEUE, N, FRONT, REAR, ITEM)

1. If FRONT = NULL OR FRONT > REAR then :
    - write: UNDERFLOW, and Return.
- Else :
    - ITEM := QUEUE[FRONT].
    - Set FRONT := FRONT + 1.

2. Return.

Circular Queue

- Drawback of linear queue
  - Once the queue is full, even though few elements from the front are deleted and some occupied space is relieved, it is not possible to add anymore new elements , as the rear has already reached the Queue's rear most position.
- Circular queue
- This queue is not linear but circular.
- Its structure can be like the following figure:
- In circular queue, once the Queue is full ,the "First" element of the queue becomes the "Rear" most element, if and only if ,the "Front" has moved forward. Otherwise it will again be a "Queue overflow" state.

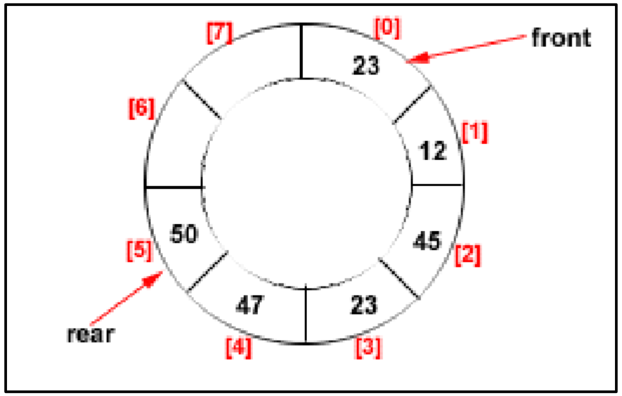


Figure: Circular Queue having Rear = 5 and Front = 0

Enqueue in circular queue

- Initially Rear = Front = -1.

1. If (Front =0 and Rear = N-1) or Front = Rear + 1

   then Print: “Circular Queue Overflow” and Return.

2. Else If Front = -1 and Rear = -1

   then Set Front := 0 and Rear := 0

3. Else If Rear = N-1 and front !=0

   then Set Rear := 0.

4. Else

   Set Rear := Rear + 1

5. Set CQueue [Rear] := Item.

6. Return

Dequeue in circular queue

1. If Front = -1 then
    - Print: “Circular Queue Underflow” and Return.
2. Set Item := CQueue [Front]
3. If Front = Rear then Set Front = Rear = -1

   Else

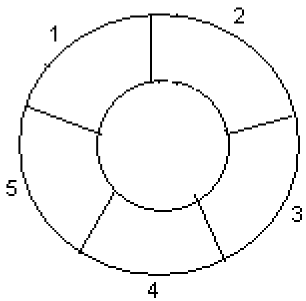
   If Front = N-1 then Set Front = 0

   Else

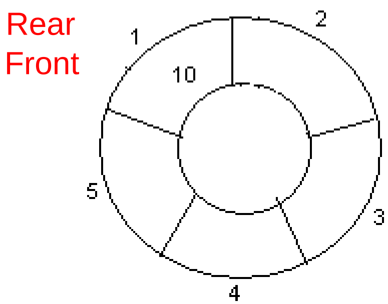
   Set Front := Front + 1
4. Return.

Example: Consider the following circular queue with N = 5.

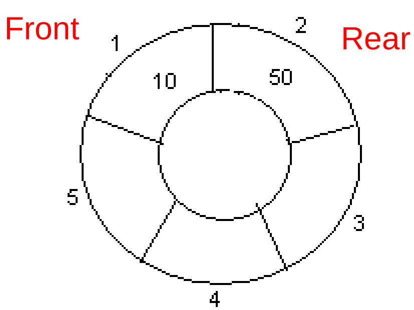
1. Initially, Rear = 0, Front = 0.



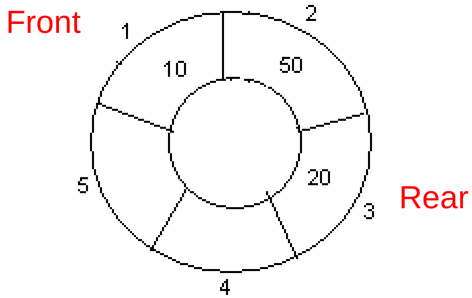
2. Insert 10, Rear = 1, Front = 1.



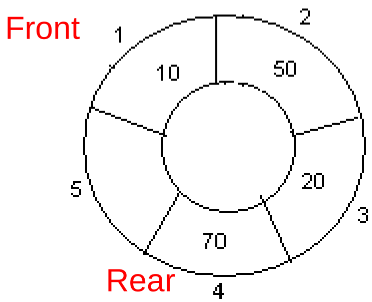
3. Insert 50, Rear = 2, Front = 1.



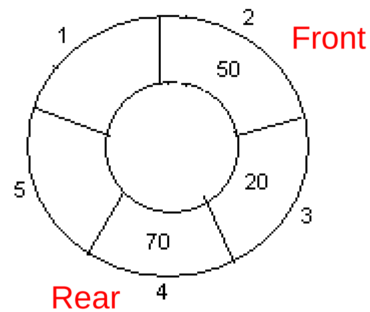
4. Insert 20, Rear = 3, Front = 1.



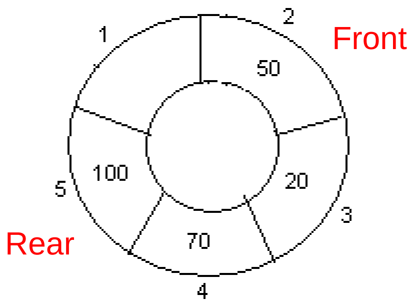
5. Insert 70, Rear = 4, Front = 1.



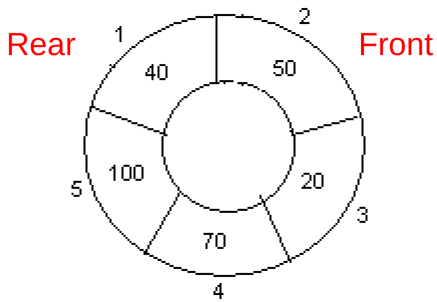
6. Delete front, Rear = 4, Front = 2.



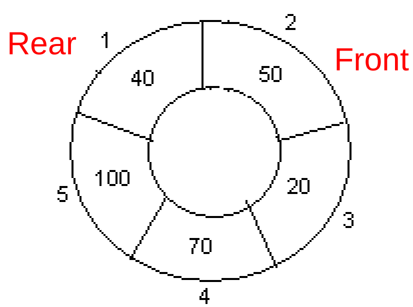
7. Insert 100, Rear = 5, Front = 2.



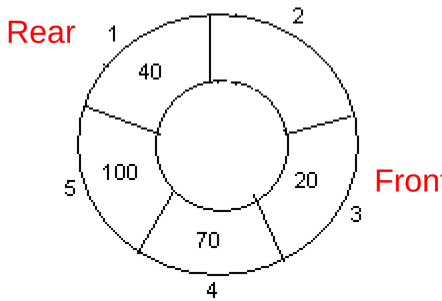
8. Insert 40, Rear = 1, Front = 2.



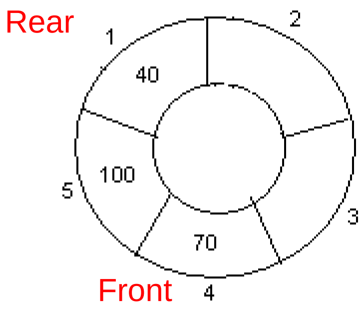
9. Insert 140, Rear = 1, Front = 2.
As Front = Rear + 1, so Queue overflow.



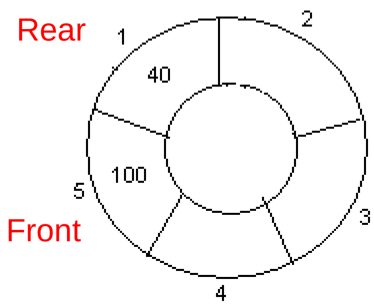
10. Delete front, Rear = 1, Front = 3.



11. Delete front, Rear = 1, Front = 4.



12. Delete front, Rear = 1, Front = 5.



Alt : By calculating position

```
qfull() {
    if(front==(rear+1)%size)
            return 1;
    else
            return 0;
    }

int insert(int item)
{
 if(qfull())
    cout<<"circular q is full";  else
 {
        if(front==-1)  front=rear=0;
     else
            rear=(rear+1)%size;
            que[rear]=item;
    }
}
```

```
int qempty()
{
  if(front==-1)
        return 1;
else
        return 0;
}

int delete()
{
        int item;
        if(qempty
        ())
            cout<<"queue is empty";
    else
    {
        item=que[front];
        if(front==rear)  {
                front=rear=-1;
        }
        else
        front=(front+1)%size;
        cout<<"the deleted item is "<<item;
•   }
```

Priority Queue

- Intrinsic ordering determines the basic operation
- Inserted and deleted based on priority
- Each element is assigned implicit or explicit priority
- If two elements have same priority, then it is processed on FCFS (First Come First Serve) basis

- Two types
  - Ascending priority queue
    - Items are inserted arbitrarily but removes smallest element first
  - Descending priority queue
    - Items are inserted arbitrarily but removes largest element first

Priority Queue

- Application
  - Scheduling queue**s** for a processor, print queues, transmit queues, backlogs, etc.….

- Representation
  - As linked list
  - Using heap
  - Multiple queues, one for each priority

Deques

- It is a double-ended queue.
- Items can be inserted and deleted from either ends.
- More versatile data structure than stack or queue.
- E.g. policy-based application (e.g. low priority go to the end, high go to the front)

- Two types:
  - Input restricted
    - Input from one end only but delete from both
  - Output restricted
    - Input from both end but delete from one end only

- Algorithm for insertion of an element at the *Rear end* of the queue:

If (Rear == MAXSIZE-1)

Queue is full

Else

Rear=Rear+1

Item=queue[Front];

Exit

- Algorithm for deletion of an element from the *front end* of the queue:

If (Front==Rear)

Queue is empty

Else

Item=queue[Front]

Front = Front+1

Exit

- Algorithm for insertion of an element at the *Front end* of the queue:

If (front==0)
  Queue is full
Else
  Front = Front-1
  Queue[Front] = item
Exit

- Algorithm for deletion of an element from the *Rear end* of the queue:

If (Front == Rear)
  Queue is empty
Else
  Rear = Rear - 1
  Item = queue[item]
Exit

THANK YOU!