Data Structures and Algorithms (CS-2001)

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY

School of Computer Engineering

4 Credit

Lecture Note

Chapter Contents

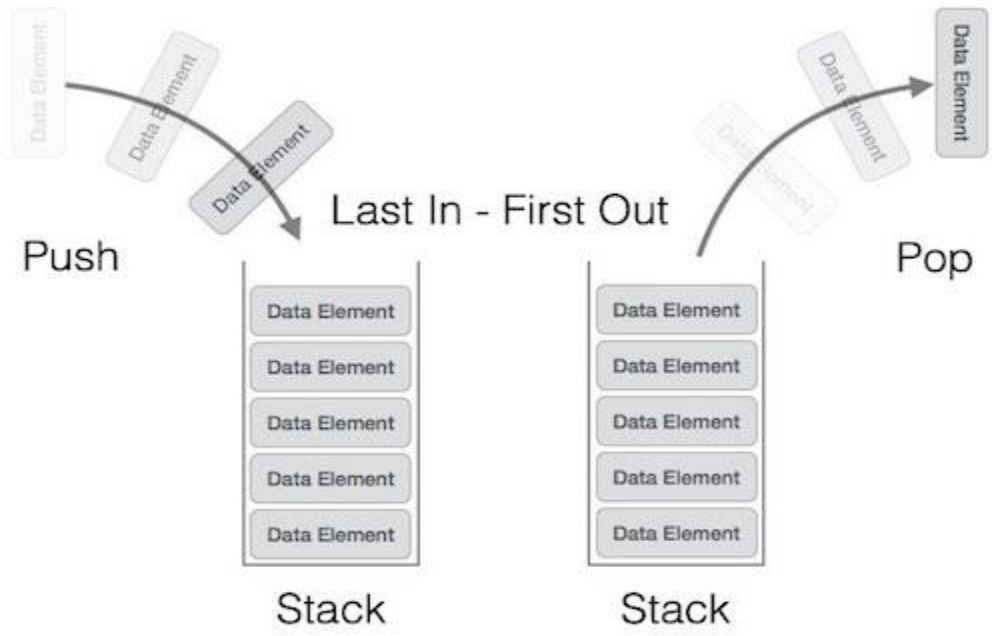
Sr #	Major and Detailed Coverage Area	Hrs
3	**Stacks and Queues**	8
	Stacks, Stacks using Dynamic Arrays and Linked Lists, Queues, Queue using Linked List, Circular Queues using Dynamic Arrays, Evaluation of Expressions, Priority Queue, Dequeue	

Introduction

The **linear lists** and **linear arrays** allowed one to insert and delete elements at any place in the list – at the beginning, at the end or in the middle. There are certain frequent situations when one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of the data structures that are useful in such situations are **stacks** and **queues**.

Stack

- A stack is a data structure that works on the principle of **Last In First Out** (LIFO).
- Last item put on the stack is the first item that can be taken off.
- New elements are added to and removed from the top called the top of the stack.



Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –

- **push** – pushing (storing) an element on the stack.
- **pop** – removing (deleting) an element from the stack.

Stack Application

- Parsing
- Recursive Function
- Function Call
- Expression Evaluation
- Expression Conversion
  - Infix to Postfix
  - Infix to Prefix
  - Postfix to Infix
  - Prefix to Infix
- Towers of Hanoi

Expression Representation

Sr#	Expression	Meaning
1	Infix	Characterized by the placement of operators between operands. E.g. plus sign in "2 + 2"
2	Prefix	Characterized by the placement of operators before operands. E.g. plus sign in "+ 2 2"
3	Postfix	Characterized by the placement of operators after operands. E.g. plus sign in " 2 2 +"

Stack Representation

Stack can be represented using –

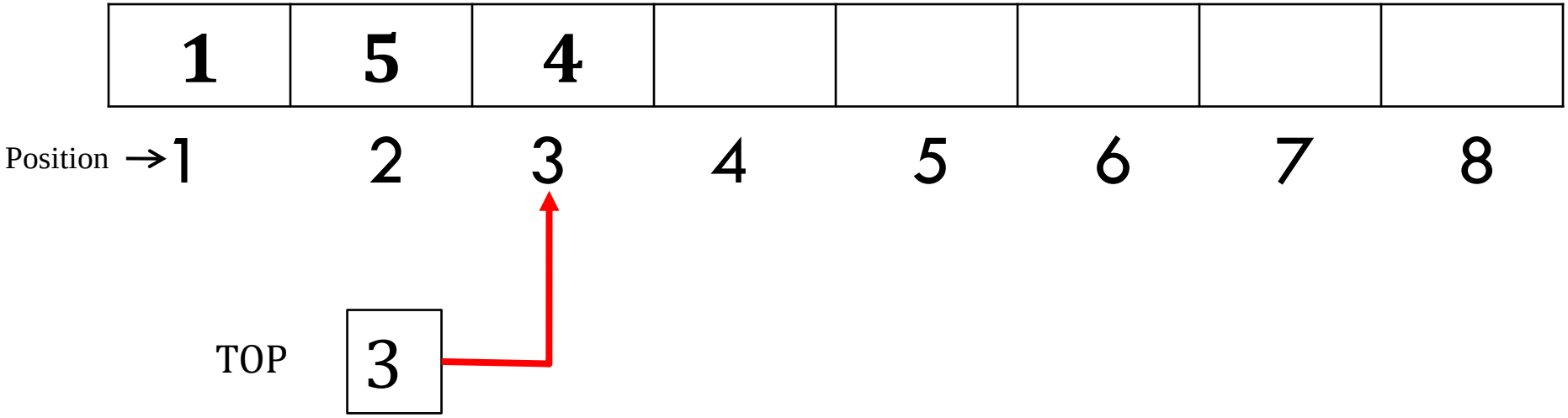
- Arrays – Storing the items contiguously
  - Fixed size (Called as Static Stack)
  - Dynamic size (Called as Dynamic Stack)
- Linked List – Storing items non-contiguously
  - Dynamic size (Called as Dynamic Stack)

Stack - Static Array Representation

Static Array Declaration & Definition:

DEFINE N NUM *[NUM represents whole number e.g. In C, equivalent statement is #define N 100]*
INTEGER STACK[N]
INTEGER TOP = 0

Pictorial Representation



TOP = 0 indicates that the stack is empty and TOP = N indicates that the stack is full

Stack Operation using Static Array

PUSH(STACK, TOP, N, ITEM)

1. Start
2. IF TOP = N THEN
   DISPLAY **Overflow**
   EXIT
   END IF
3. TOP = TOP + 1
4. STACK[TOP] = ITEM *[Insert Item into new TOP Position]*
5. Stop

POP(STACK, TOP, ITEM)

1. Start
2. IF TOP = 0 THEN
   DISPLAY **Underflow**
   EXIT
   END IF
3. SET ITEM = STACK[TOP] *[Call by reference]*
4. TOP = TOP -1
5. Stop

PEEK(STACK, TOP, ITEM)

1. Start
2. IF TOP = 0 THEN
   DISPLAY **Underflow**
   EXIT
   END IF
3. SET ITEM = STACK[TOP] *[Call by reference]*
4. Stop

Operation Description

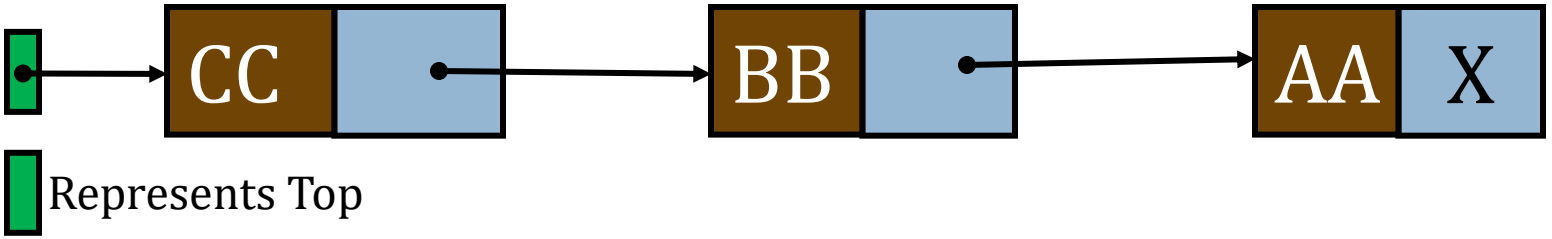
- **push** – pushing (storing) an element on the stack.
- **pop** – removing (deleting) an element from the stack
- **Peek** - get the top data element of the stack, without removing it.

Class Work

1. Design a Push algorithm using dynamic array
2. Design a Pop algorithm using dynamic array
3. Design an algorithm to realize two stacks in single static array
4. Design an recursive algorithm to reverse the stack

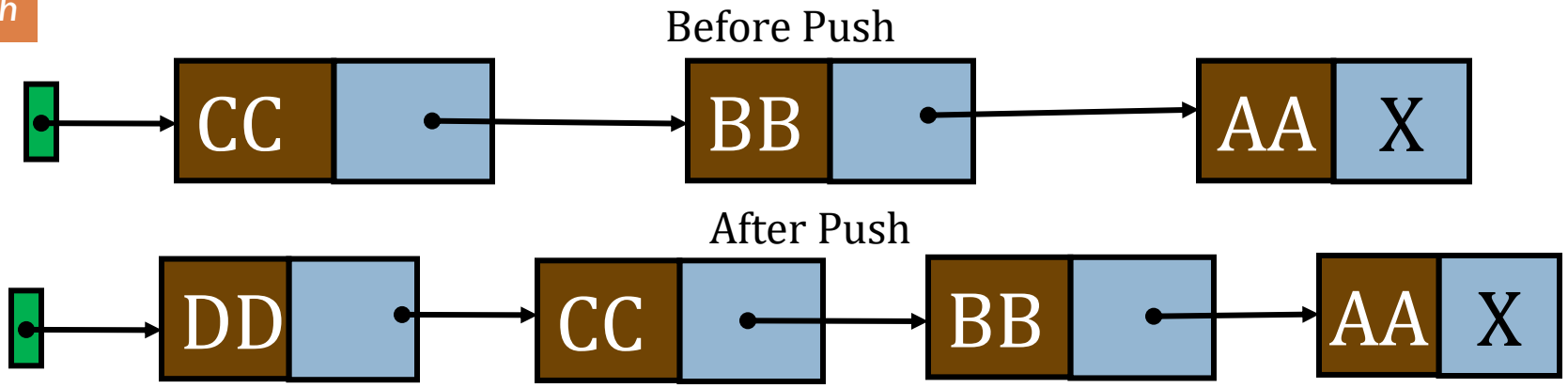
Stack - Linked List Representation

Since all the action happens at the top of the stack, a **single linked list** is a fine way to represent the stack wherein the header/start always points to the top of the stack.



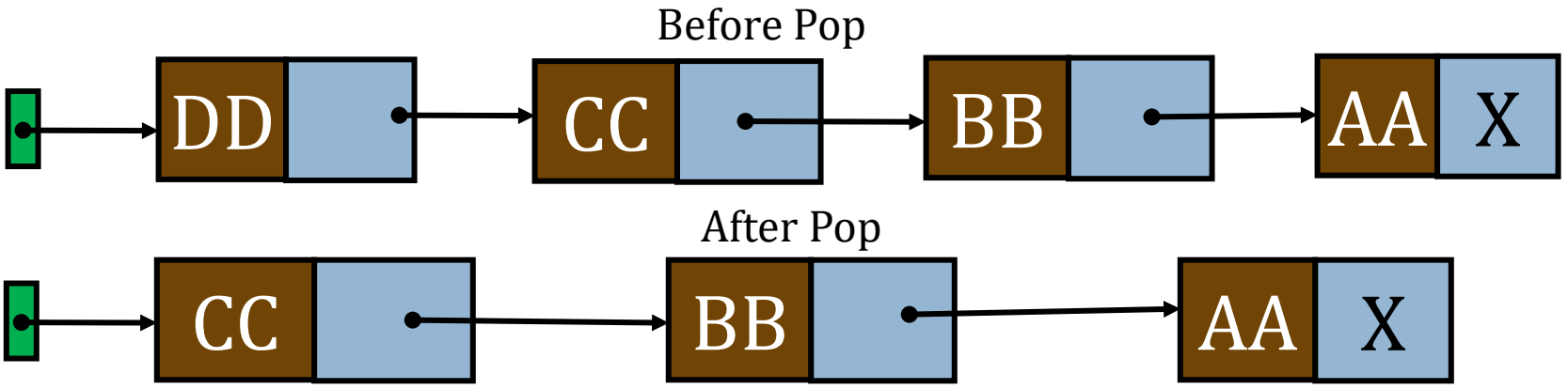
- **Pushing** is inserting an element at the front of the list
- **Popping** is removing an element from the front of the list

Push



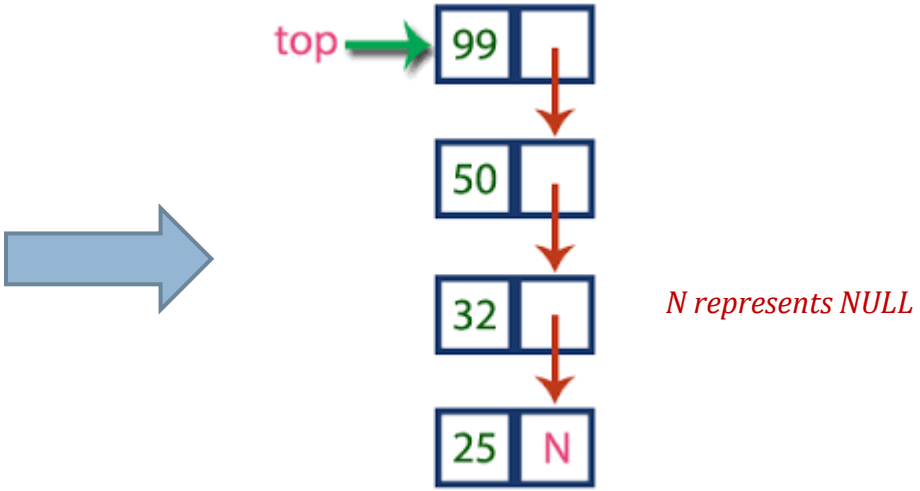
Stack - Linked List Representation

Pop



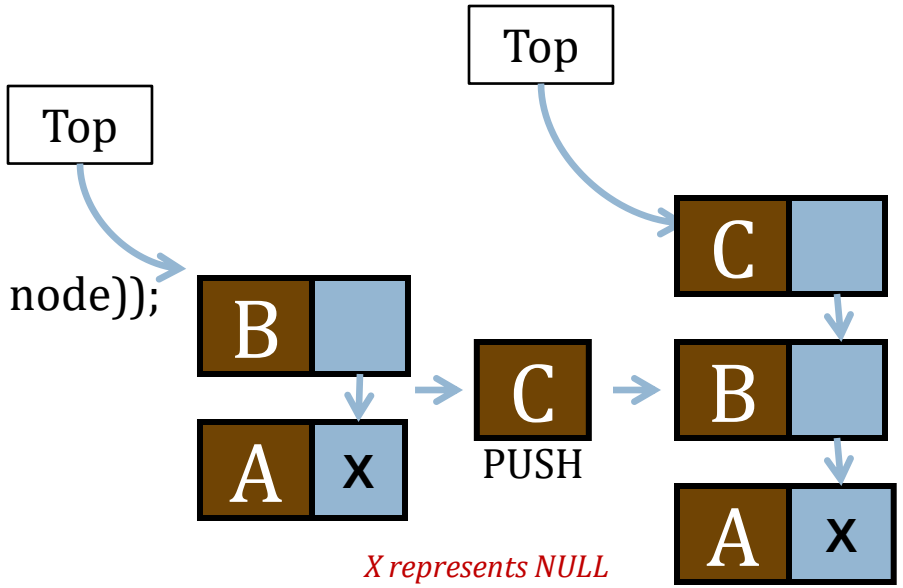
Stack Node Representation

```
struct node
{
      int info;
      struct node *next;
}*top;
```



Push Function


```
void push()
{
  int data;
  scanf("%d", &data);
 if (top == NULL)
  {
    top =(struct node *)malloc(*sizeof(struct node));
    top->next = NULL;
    top->info = data;
 }
 else
 {
     struct node * temp =(struct node *)malloc(*sizeof(struct node));
     temp->next = top;
     temp->info = data;
     top = temp;
 }
```

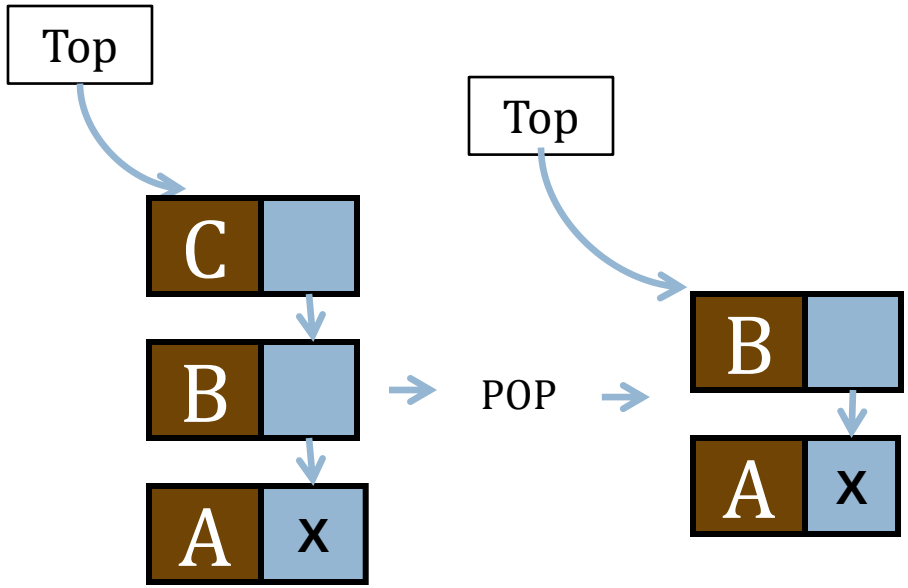


Pop Function


```
void pop()
{
  struct node *tempTop = top;

   if (tempTop == NULL)
   {
      printf("\n Error ");
      return;
   }
   else
      tempTop = tempTop ->next;

   printf("\n Popped value : %d", top->info);
   free(top);
   top = tempTop;
```



X represents NULL

Arithmetic Expression : Polish Notation

The way to write arithmetic expression is known as a notation. **Polish Notation** is a way of **expressing arithmetic expressions** that **avoids the use of brackets** to define priorities for evaluation of operators. Polish Notation was devised by the Polish philosopher and mathematician Jan Łukasiewicz (1878-1956) for use in symbolic logic.

- Evaluate the following parenthesis-free arithmetic expression

2 î 3 + 5 * 2 î 2 – 12 / 6

= 8 + 5 * 4 – 12 / 6 = 8 + 20 – 2 = 26

An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are –

- Infix Notation
- Postfix Notation
- Prefix Notation

Expression Representation

- **Infix Notation** - We write expression in infix notation, e.g. a - b + c, where operators are used in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.
- **Prefix Notation** - In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, +ab. This is equivalent to its infix notation a + b. Prefix notation is also known as **Polish Notation**.
- **Postfix Notation** - This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, ab+. This is equivalent to its infix notation a + b.

Difference in 3 Notation

Example

Sr#	Infix	Prefix	Postfix
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

Parsing Expression

Parsing is the process of analyzing a collection of symbols, either in natural language or in computer languages, conforming to the rules of a formal grammar.

It is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed. To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence :- When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example – a + b *c ➔ a + (b * c)

Associativity :- It describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as (a + b) − c.

Parsing Expression cont...

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) –

Sr#	Operator	Precedence	Associativity
1	Exponentiation (^ or î)	Highest	Right Associative
2	Multiplication (∗), Division (/) and Modular Division (%)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis.

For example – In a + b*c, the expression part b*c will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for a + b to be evaluated first, like (a + b)*c.

Arithmetic Expression Conversion

Infix expression to Prefix notation

(A+B)*C = [+AB]*C = *+ABC where *[ ] indicates a partial translation*

A+(B*C) = A+[*BC] = +A*BC

(A+B)/(C-D) = [+AB]/[-CD] = /+AB-CD

Infix expression to Postfix notation

(A+B)*C = [AB+]*C = AB+C* where *[ ] indicates a partial translation*

A+(B*C) = A+[BC*] = ABC*+

(A+B)/(C-D) = [AB+]/[CD-] = AB+CD-/

Note

Order in which the operations are to be performed is completely determined by the positions of the operators and operand in the expression. Accordingly, one never needs parenthesis when writing expression in Polish notations

Why Postfix and Prefix?

Postfix:

Computer usually evaluates an arithmetic expression written in infix notation in two steps:

- **1st Step -** Converts the infix notation to Postfix notation
- **2nd Step -** Evaluates the Postfix expression.

In each step, stack is the main tool used to accomplish the given task.

Prefix:

- Has seen wide application in Lisp (programming language) s-expressions
- Data Compression with Prefix Encoding
- Standard scientific prefixes are used to denote powers of 10 e.g. kilo $= 1e^3$
  - mega $= 1e^6$
  - giga $= 1e^9$
  - tera $= 1e^{12}$
  - peta $= 1e^{15}$
  - exa $= 1e^{18}$

Evaluation of a Postfix Expression

1. Start
2. Validate the Postfix expression for correctness
   a) '(' and ')' are in pairs
   b) Operator is after the operands *[Binary operators are considered only]*
3. Add a right parenthesis ')" at the end of P. *[This act as delimiter]*
4. Scan P from left to right and repeat steps 5 and 6 for each element of P until the delimiter ")" is encountered
5. If an operand is encountered, put it on STACK
6. 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 step (b) on STACK
7. Set value of the postfix expression equal to the top element of STACK
8. Stop

Example

- P = 5, 6, 2, + , *, 12, 4, /, - [Postfix]

Note – Commas are used to separate the elements of P so that 5,6,2 is not interpreted as 562.

P: *(after adding a sentinel right parenthesis at the end of P)*

5 6 2 + * 12 4 / -)

(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)

Symbol Scanned		STACK		
(1)	5	5		
(2)	6	5	6	
(3)	2	5	6	2
(4)	+	5	8	
(5)	*	40		
(6)	12	40	12	

Symbol Scanned		STACK		
(7)	4	40	12	4
(8)	/	40	3	
(9)	-	37		
(10)	)			

Q = 5 * (6 + 2) – 12 / 4 is the equivalant Infix expression

Class Work

Consider the following arithmetic expression P, written in postfix notation:
P =12, 7 , 3, - , / , 2, 1, 5, +, *, +

1. Evaluate the above postfix expression
2. By inspection, translate P into its equivalent infix expression
3. Evaluate the converted infix expression

Q = 12 / (7-3) + 2 * (1+5) = 15

Evaluation of a Prefix Expression

1. Start
2. Validate the Prefix Expression for correctness
   a) '(' and ')' are in pairs
   b) Operator is before the operands *[Binary operators are considered only]*
3. Scan P from right to left and repeat steps 5 and 6 for each element of P until it ends.
4. If an operand is encountered, put it on STACK
5. 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 A ⊘ B
   (c) Place the result of step (b) on STACK
6. Set value of the prefix expression equal to the top element of STACK
7. Stop

Example

P = -, *, +, 4, 3, 2, 5 [Prefix]

Note – Commas are used to separate the elements of P

-	*	+	4	3	2	5
(7)	(6)	(5)	(4)	(3)	(2)	(1)

Symbol scanned		STACK			
(1)	5	5			
(2)	2	5	2		
(3)	3	5	2	3	
(4)	4	5	2	3	4
(5)	+	5	2	7	
(6)	*	5	14		
(7)	-	9			

Infix to Postfix Conversion

Q is an arithmetic expression written in infix notation. This algorithm **InfixToPostfix (Q, P)** finds the equivalent postfix notation where P is the equivalent postfix notation –

1. Start
2. Validate the Infix expression for correctness
   a) '(' and ')' are in pairs
   b) Operator is between the operands *[Binary operators are considered only]*
3. Push "(" onto STACK and ")" to the end of Q
4. Scan Q from Left to Right and Repeat Steps 5 to 8 for each element of Q until the STACK is empty
5. If an operand is encountered, add it to P
6. If a left parenthesis is encountered, push it onto STACK
7. If an operator ⊘ is encountered, then:
   (a) Repeatedly pop from STACK and add to P each operator which has same precedence or higher precedence than ⊘.
   (b) Add ⊘ to STACK.
8. If a right parenthesis is encountered, then
   (a) Repeatedly pop from the STACK and add to P each operator until a left parenthesis is encountered.
   (b) Remove the left parenthesis. *[Do not add it to P]*
9. Stop

Example

Q : A + (B * C – (D / E î F) * G) * H

So first, we push "(" onto stack, and then add ")" to the end of the Q to obtain:

A	+	(	B	*	C	-	(	D	/	E	î	F	)	*	G	)	*	H	)
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

	Symbol Scanned	STACK	Expression P
1	A	(	A
2	+	(+	A
3	(	(+ (	A
4	B	(+ (	A B
5	*	(+ (*	A B
6	C	(+ (*	A B C
7	-	(+ (-	A B C *

Example cont...

A	+	(	B	*	C	-	(	D	/	E	î	F	)	*	G	)	*	H	)
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

	Symbol Scanned	STACK	Expression P
8	(	(+ (- (	A B C *
9	D	(+ (- (	A B C * D
10	/	(+ (- (/	A B C * D
11	E	(+ (- (/	A B C * D E
12	î	(+ (- (/ î	A B C * D E
13	F	(+ (- (/ î	A B C * D E F
14	)	(+ (-	A B C * D E F î /
15	*	(+ (- *	A B C * D E F î /
16	G	(+ (- *	A B C * D E F î /G
17	)	(+	A B C * D E F î / G * -
18	*	(+ *	A B C * D E F î / G * -
19	H	(+ *	A B C * D E F î / G * - H
20	)		**A B C * D E F î / G * - H * +**

Class Work

Consider the infix expression Q: ((A + B) * D) î (E-F). Translate Q into its equivalent postfix expression P

	Symbol Scanned	STACK	Expression P
1	(	((	
2	(	(((	
3	A	(((	A
4	+	(((+	A
5	B	(((+	A B
6	)	((	A B +
7	*	((*	A B +
8	D	((*	A B + D
9	)	(	A B + D *
10	î	(î	A B + D *
11	(	(î (	A B + D *
12	E	(î (	A B + D * E
13	-	(î (-	A B + D * E F
14	F	(î (-	A B + D * E F
15	)	(î	A B + D * E F –
16			A B + D * E F – î

Infix to Prefix Conversion

Q is an arithmetic expression written in infix notation. This algorithm **InfixToPrefix(Q, P)** finds the equivalent postfix notation where P is the equivalent prefix notation –

1. Start
2. Validate the Infix expression for correctness
   a) '(' and ')' are in pairs
   b) Operator is between the operands [Binary operators are considered only]
3. Reverse the expression
4. **CALL** InfixToPostfix (Q, P)
5. Reverse the expression
6. Stop

Example: Infix notation: A+B*C

Step 1 – Infix expression is correct

Step 2 – Reversing the expression resulting to C*B+A

Step 3 – Postfix expression produced in step 2 is CB*A+

Step 4 – Reversing the expression produced in step 3 is **+ A * B C**

Result: + A * B C

Example cont…

Q : (A+B î C)*D+E î F

Step 1 – Infix expression is correct

Step 2 –

a) Reversing the expression produce F î E + D *) C î B + A (

b) Reverse the bracket, so making every '(' as ')' and every ')' as '(' revised expression is F î E + D * (C î B + A)

Step 3 – Postfix expression of step 2 produce F E î D C B î A + * +

Step 4 – Reversing the expression produced in step 3 becomes

+ * + A î B C D î E 5

Result: + * + A î B C D î E 5

Home Work

Read the following at your own pace:

- Postfix to Infix conversion – http://scanftree.com/Data_Structure/postfix-to-infix
- Prefix to Infix conversion – http://scanftree.com/Data_Structure/prefix-to-infix
- Postfix to Prefix conversion – http://see-programming.blogspot.in/2013/05/postfix-to-prefix-conversion.html
- Postfix to Prefix conversion – http://www.manojagarwal.co.in/conversion-from-prefix-to-postfix/

Queue

- A queue is a data structure that models/enforces the first-come first-serve order, or equivalently the first-in first-out (**FIFO**) order.
- The element that is inserted first into the queue will be the element that will deleted first, and the element that is inserted last is deleted last.
- Items are inserted one end (rear end) and deleted from the other end(front end).

Queue basic operations are –

- **enqueue**– insert element at rear
- **dequeue**– Remove element from front
- **peek –** Examine the element at front without removing it from queue

Queue Application

Operating System

- queue of print jobs to send to the printer
- queue of programs / processes to be run
- queue of network data packets to send

Programming

- modeling a line of customers or clients
- storing a queue of computations to be performed in order

Real World

- people on an escalator or waiting in a line
- cars at a gas station (or on an assembly line)

Queue Representation

Queue can be represented using –

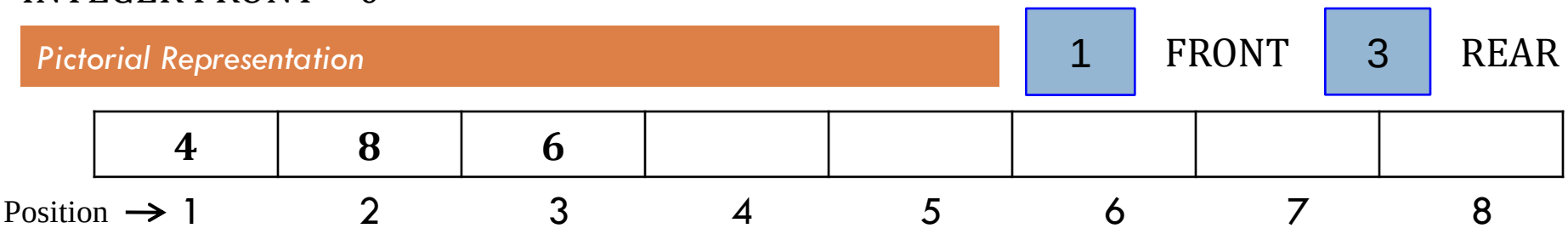
- Static Arrays – Storing the items contiguously
  - Static Queue due to fixed size
  - Space is wasted if we use less elements
  - cannot insert more elements than the array can hold
- Dynamic Arrays – Storing the items contiguously
  - Dynamic Queue
- Linked List– Storing items non-contiguously
  - Dynamic Queue

Queue - Static Array Representation

Static Array Declaration & Definition:

DEFINE N NUM *[NUM represents whole number e.g. In C, equivalent statement is #define N 100]*
INTEGER QUEUE[N]
INTEGER REAR = 0
INTEGER FRONT = 0

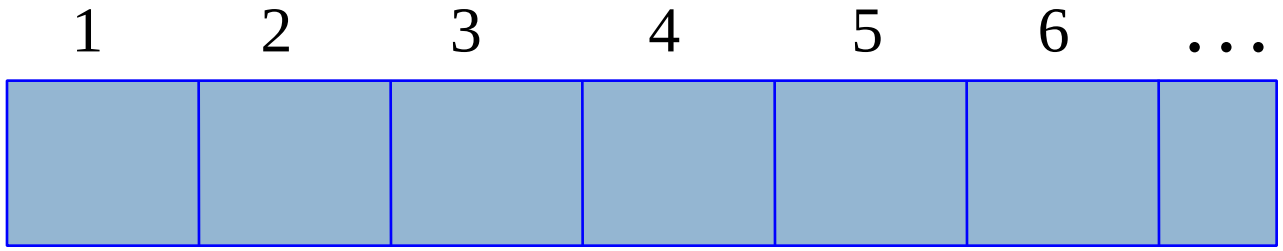
Pictorial Representation



Keeps track of the number of items in the queue and the position of the **front** element (at the front of the queue), the **rear** element (at the rear). The position of the front element is stored in the **front** member variable. The next item is after the first one and so on until the rear of the queue that occurs at the index stored in a member variable called **rear**.

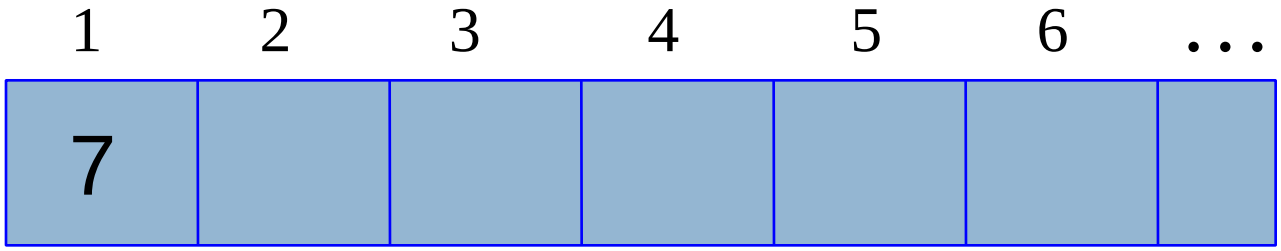
FRONT = 0 and REAR = 0 indicates that the queue is empty and REAR = N indicates that the queue is full

Enqueue Operation



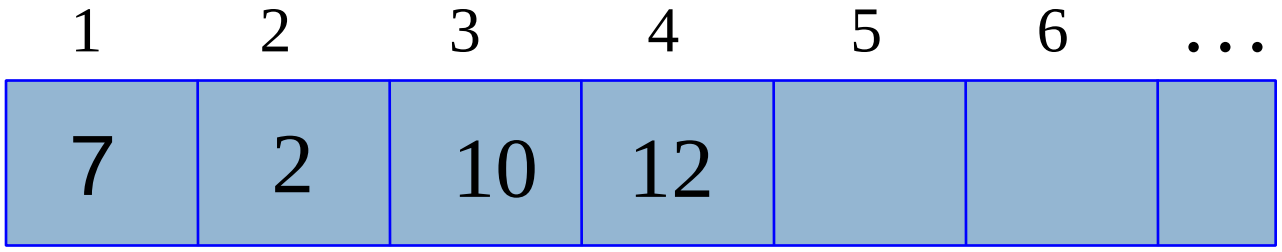
0	FRONT
0	REAR

Let's an element enters the queue…



1	FRONT
1	REAR

After a series of insertion, the state of the queue is



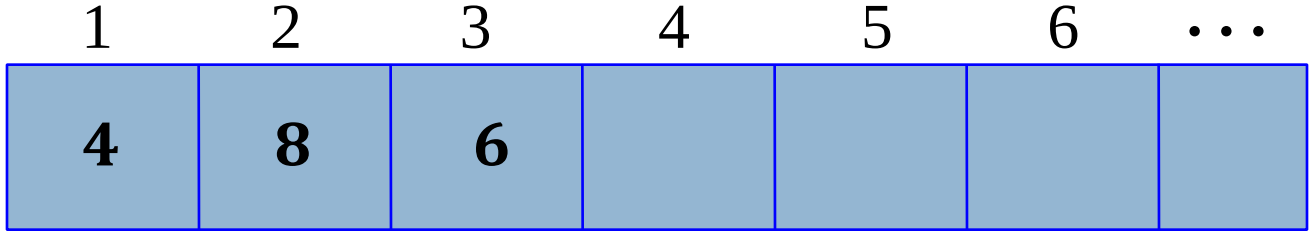
1	FRONT
4	REAR

Enqueue Algorithm

SQINSERT(QUEUE, N, FRONT, REAR, ITEM)

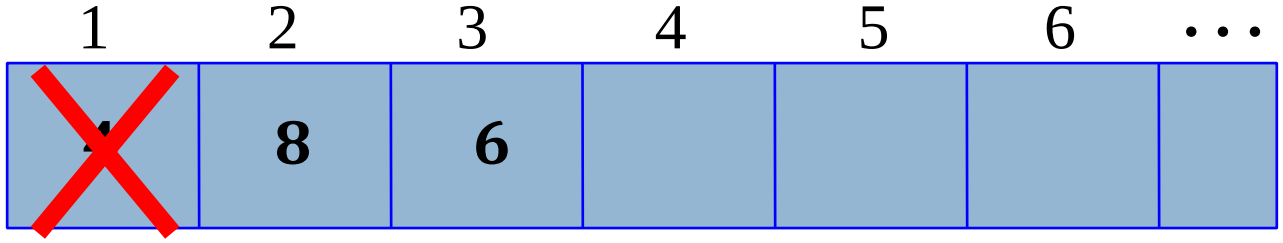
1. Start
2. IF (REAR = N) THEN
   DISPLAY "OVERFLOW"
   EXIT
   END IF
3. IF (FRONT = 0 AND REAR = 0) THEN *[Queue initially empty]*
   FRONT = 1
   REAR = 1
   ELSE
   REAR = REAR + 1
   END IF
4. QUEUE[REAR] = ITEM *[This inserts new element]*
5. Stop

Dequeue Operation



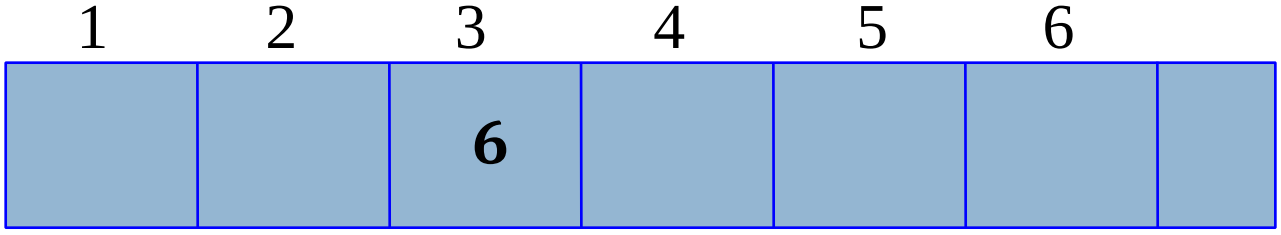
1	FRONT
3	REAR

Let's an element leaves the queue



2	FRONT
3	REAR

After a series of deletion , the state of the queue is



3	FRONT
3	REAR

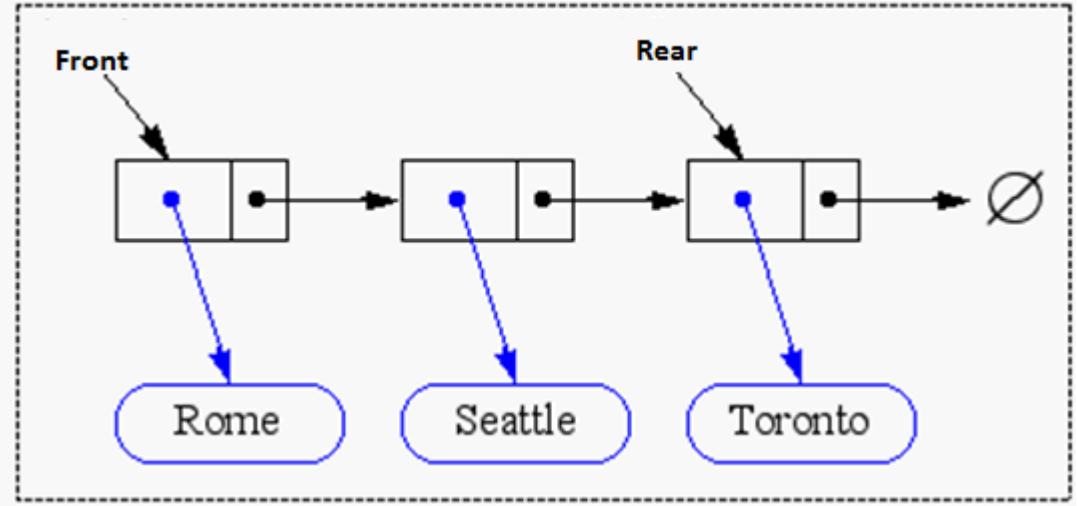
Dequeue Algorithm

ITEM SQDELETE(QUEUE, FRONT, REAR)

1. Start
2. IF FRONT = 0 THEN
   DISPLAY "UNDERFLOW"
   EXIT
   END IF
3. ITEM = QUEUE[FRONT]
4. IF (FRONT == REAR) THEN *[Check if only one element is left]*
   FRONT = 1
   REAR = 1
   ELSE
   FRONT = FRONT + 1 *[Increment FRONT by 1]*
   END IF
5. RETURN ITEM
6. Stop

Queue Linked Representation

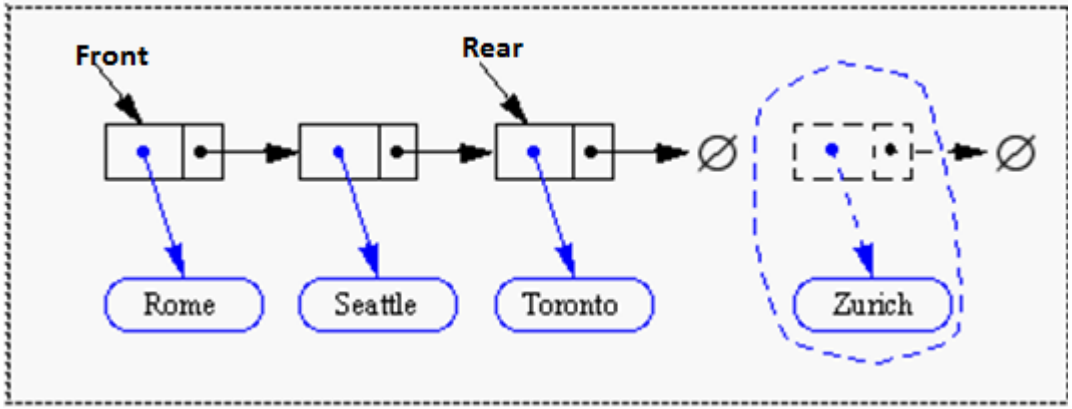
Nodes connected in a chain by links



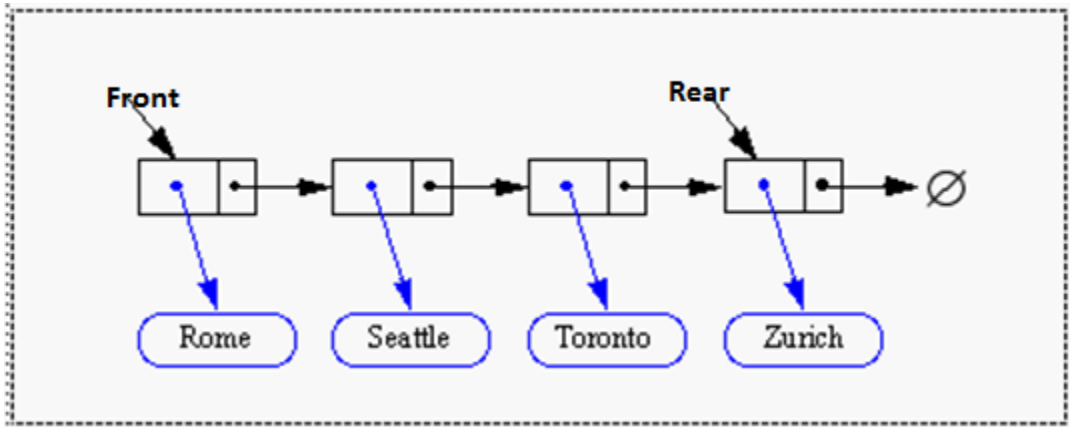
Node Representation

```
struct node
{
      char info[50];
      struct node *next;
}*front, *rear;
```

Enqueue Operation



Post Insertion

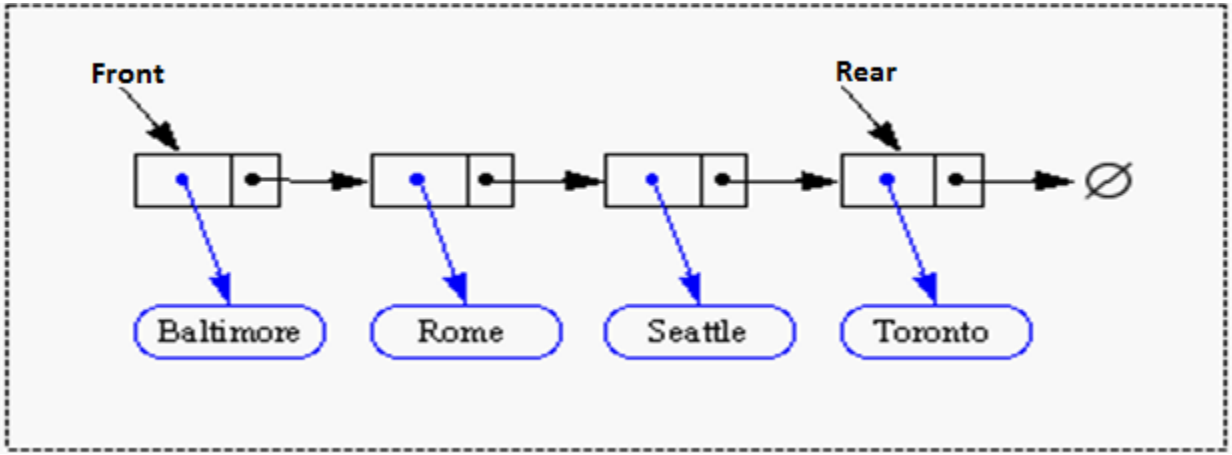


Enqueue Algorithm

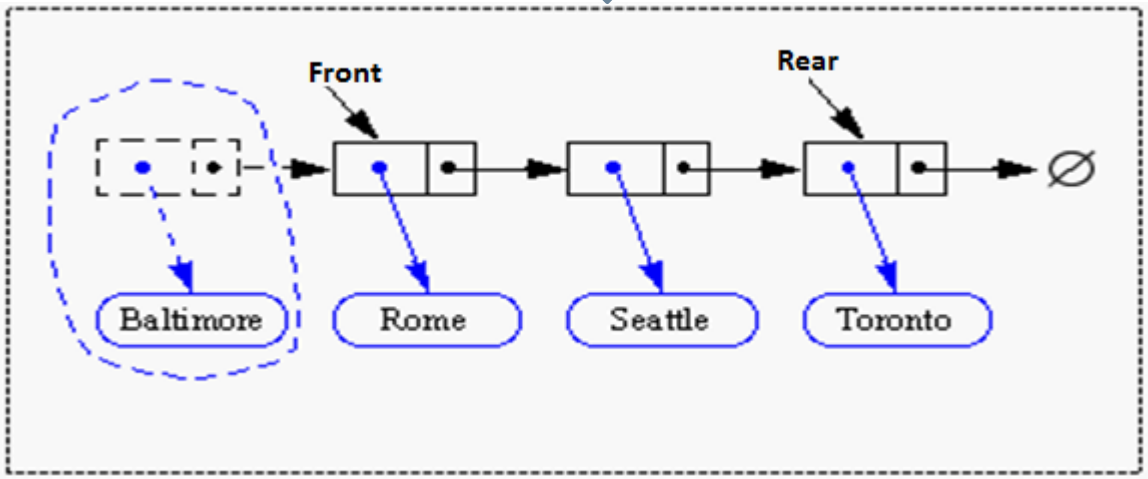
*LLQINSERT(struct node * FRONT, struct node * REAR, ITEM)*

```
BEGIN
    struct list* node= (struct node*)malloc(sizeof(struct node));
    node->info =  ITEM;
    node -> next = NULL;
   IF  (FRONT ==NULL) THEN
       FRONT = node
       REAR = node
  ELSE
       REAR ->next = node
       REAR = node
   END IF
END
```

Dequeue Operation



Post Deletion



Dequeue Algorithm

*ITEM LLQDELETE(struct node * FRONT)*

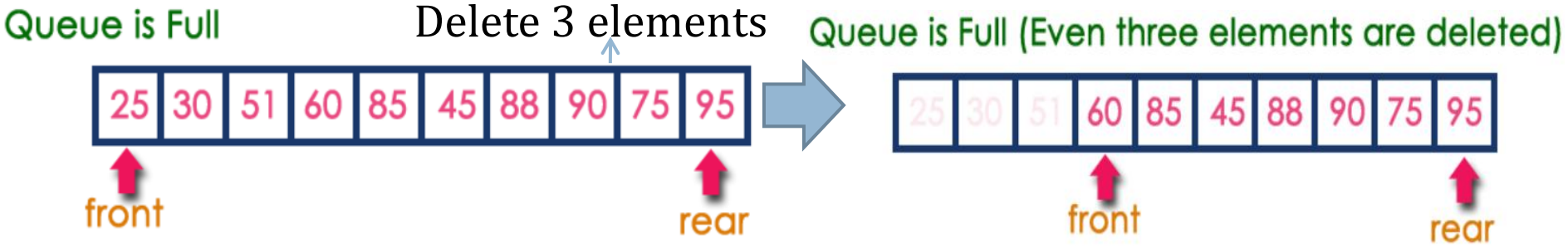
```
BEGIN
   IF (FRONT = NULL) THEN
      DISPLAY “Queue Underflow"
      EXIT
   END IF
   struct * node temp = FRONT
   ITEM = temp->info
   FRONT = FRONT->next
   FREE(temp)
   RETURN ITEM
END
```

Class Work

1. Design a Deque algorithm using dynamic array
2. Design an Enqueue algorithm using dynamic array

Linear Queue Drawback

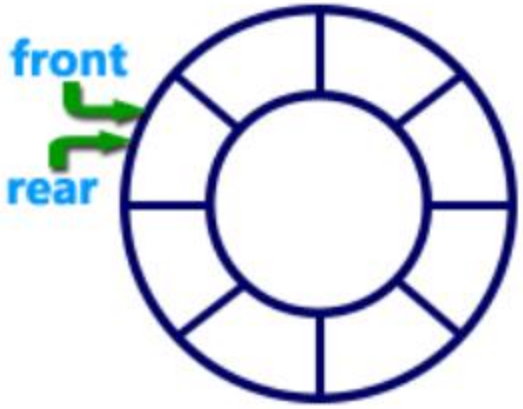
Once the linear queue is full, even though few elements from the front are deleted & some occupied space is relieved, it is not possible to add anymore new elements, as the rear has already reached the queue's rear most position.



This situation also says that queue is full and we can not insert the new element because, 'rear' is still at last position. In above situation, even though we have empty positions in the queue we can not make use of them to insert new element. This is the major problem in normal queue data structure. To overcome this problem we use **circular queue** data structure.

Circular Queue

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle and the graphical representation of a circular queue is as follows...



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.

Circular Queue Enqueue Algorithm

CQINSERT(QUEUE, N, FRONT, REAR, ITEM)

1. Start
2. IF (FRONT = 0 AND REAR = 0) THEN
   FRONT = 1
   GOTO STEP 5
   END IF
3. IF (FRONT =1 AND REAR = N) OR (FRONT = REAR + 1) THEN
   DISPLAY "Circular Queue Overflow"
   EXIT
   END IF
4. IF (REAR = N) THEN
   REAR = 1
   GOTO STEP 6
   END IF
5. REAR = REAR + 1
6. QUEUE [REAR] = ITEM
7. Stop

Circular Queue Dequeue Algorithm

ITEM CQDELETE(QUEUE, N, FRONT, REAR)

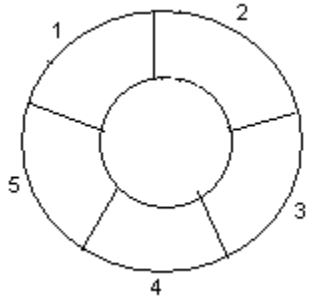
```
1. Start
2. IF (FRONT = 0) THEN
        DISPLAY "Circular Queue Underflow"
        EXIT
   END IF
3.  ITEM = QUEUE[FRONT]
4. IF FRONT = N THEN
      FRONT = 1
      RETRUN ITEM
   END IF
```

[Continuation of algorithm]

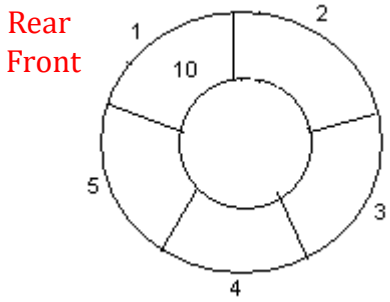
```
5. IF (FRONT = REAR) THEN
      FRONT = 0
      REAR = 0
      RETURN ITEM
   END IF
6. FRONT = FRONT + 1
7. RETURN ITEM
8. Stop
```

Circular Queue Example

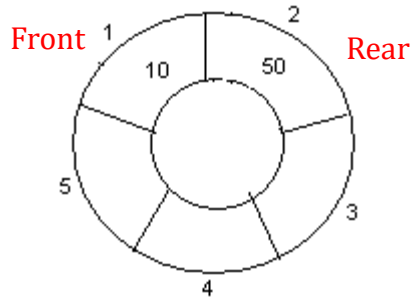
1. Initially, Rear = 0, Front = 0.



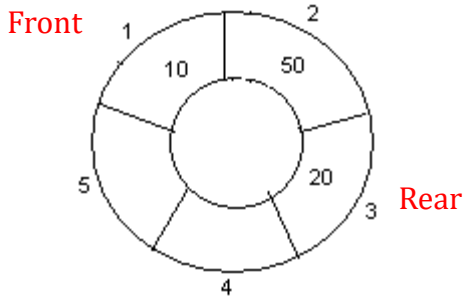
2. Insert 10, Rear = 1, Front = 1.



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

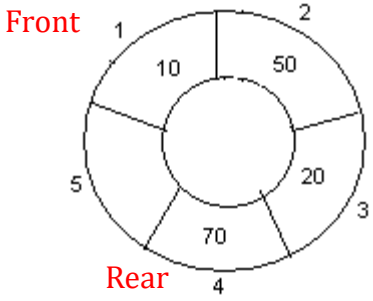


4. Insert 20, Rear = 3, Front = 0.

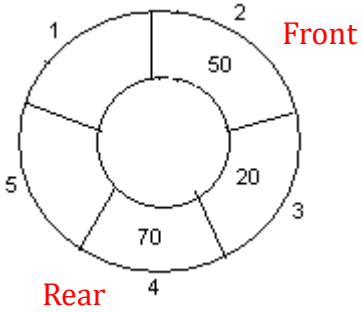


Circular Queue Example cont…

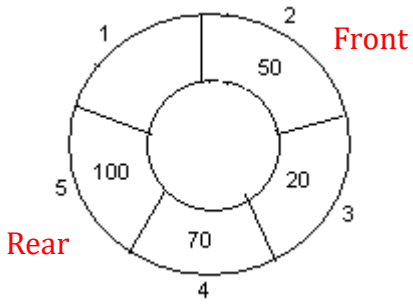
5. Insert 70, Rear = 4, Front = 1.



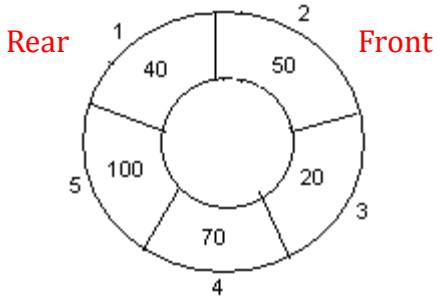
6. Delete front, Rear = 4, Front = 2.



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



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

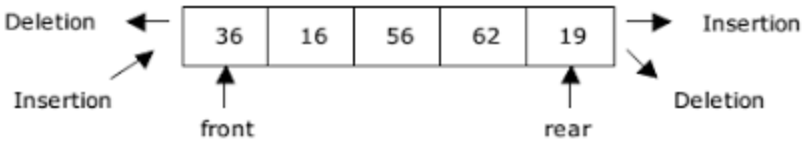


Class Work - Circular Queue

1. Design an **Enqueue** algorithm using dynamic array
2. Design a **Dequeue** algorithm using dynamic array
3. Design an **Enqueue** algorithm using Linked List
4. Design a **Dequeue** algorithm using Linked List

Deques

- A deque is a **d**ouble-ended **q**ueue
- Insertions and deletions can occur at either end, but not in the middle
- Implementation is similar to that for queues
- Deques are not heavily used
- You should know what a deque is, but we won't explore them much further
- There are 2 types of deque as explain below -



Input-restricted Deque

- Elements can be inserted only at one end.
- Elements can be removed from both the ends

Output-restricted Deque

- Elements can be removed only at one end.
- Elements can be inserted from both the ends.

Deques Construction

The Dequeue can be constructed in two ways and are:

1) Using Array 2) Using Linked List

Static Arrays: FRONT is initialized to 0, REAR is initialized to N where N represents array length

ENQUE_FRONT(QUEUE, FRONT, REAR, ITEM)

1. Start
2. IF (FRONT = REAR) *[QUEUE is Full]*
   Display "Over flow"
   EXIT
   END IF
3. FRONT = FRONT + 1
4. QUEUE[FRONT] = ITEM
5. Stop

ENQUE_REAR(QUEUE, FRONT, REAR, ITEM)

1. Start
2. IF (REAR = FRONT) *[QUEUE is Full]*
   Display "Over flow"
   EXIT
   END IF
3. QUEUE[REAR] = ITEM
4. REAR = REAR - 1
5. Stop

Deques Construction cont…

ITEM DEQUE_FRONT(QUEUE, FRONT, REAR)

1. Start
2. IF (FRONT = 0) *[QUEUE is Empty]*
   DISPLAY "Under flow"
   EXIT
   END IF
3. ITEM = QUEUE[FRONT]
4. FRONT = FRONT - 1
5. RETURN ITEM
6. Stop

ITEM DEQUE_REAR(QUEUE, N, FRONT, REAR)

1. Start
2. IF (REAR = N) *[QUEUE is Empty]*
   DISPLAY"Under flow"
   EXIT
   END IF
4. ITEM = QUEUE[REAR]
5. REAR = REAR + 1
6. RETURN ITEM
7. Stop

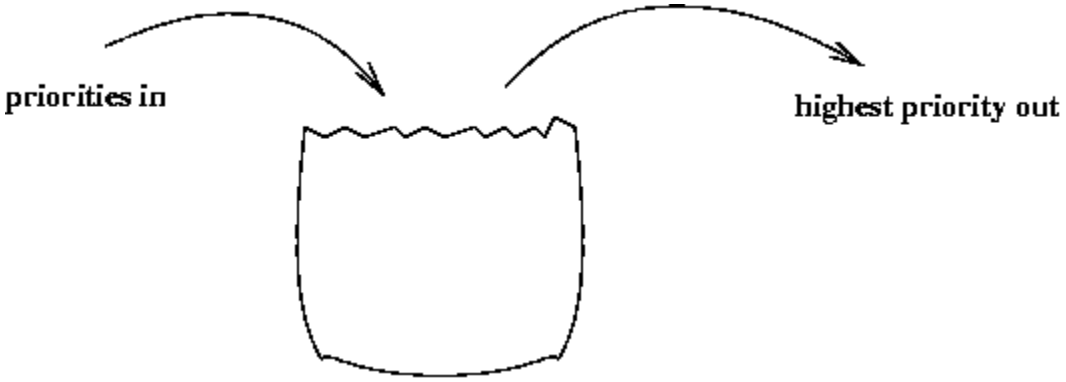
Class Work - Deques

1. Design an **Enqueue** algorithm using dynamic array
2. Design a **Dequeue** algorithm using dynamic array
3. Design an **Enqueue** algorithm for using Linked List
4. Design a **Dequeue** algorithm using Linked List

Priority Queues

A priority queue is a collection of elements such that each element has been assigned a **priority** and such that the order in which elements are deleted and processed comes from the following rules –

- An element of higher priority is processed before any element of lower priority
- Two elements with the priority are processed according to the order in which they were added to the queue.



Maintaining priority queues are –

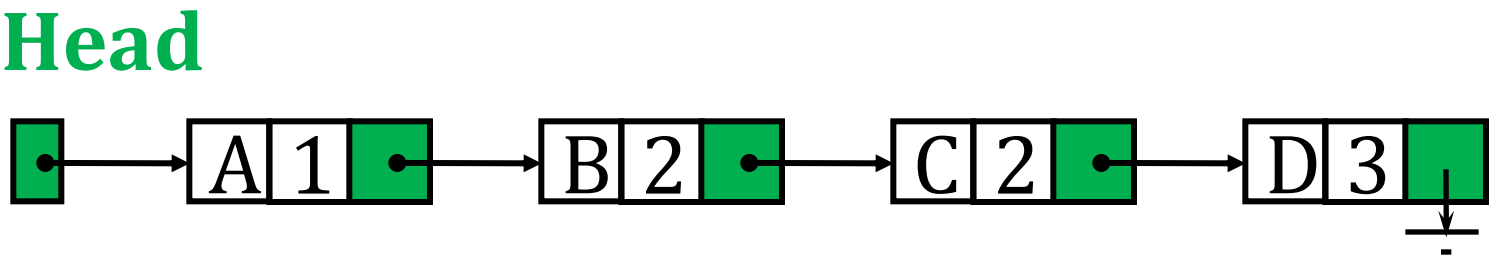
- One-Way List
- Array

Priority Queue - One-Way List Representation

- Each node in the list will contain three items of information: an information field INFO, a priority number PRN and a link number LINK.

- A node X precedes a node Y in the list when:
  - X has higher priority than Y or
  - When both have the same priority but X was added to the list before Y.

Systematic diagram



Priority Queue - One-Way List Representation

Node Representation

```
struct node
{
      int info;
      int prn;
      struct node *next;
}*top;
```

Deletion

1. Start
2. Visit to the second last node in the chain
3. Make the second last node next field to NULL
4. Free the last node
6. Stop

Insertion with Priority N and ITEM

1. Start
2. Traverse the list until finding a node X whose priority number exceeds the priority N.
3. If node found then Insert ITEM in front of node X
4. If no such node is found, insert ITEM as the last item of the list
5. Stop

Priority Queue - Array Representation

- Another way to maintain a priority queue in memory is to use a separate queue for each priority number.
- Each such queue will appear in its **own circular array** and must have its own pair of pointers, FRONT and REAR
- If each queue is allocated the same amount of space, a **two–dimensional array** QUEUE can be used instead of the linear array
- FRONT[K] and REAR[K] contains the front and rear elements of row K of queue, the row that maintains the queue of elements with priority number K

Systematic diagram

Priority	1	2	3	4	5
1		A			
2	B	C	G		
3					
4	E				D
5				F	

	FRONT	REAR
1	2	2
2	1	3
3	0	0
4	5	1
5	4	4

Priority Queue – Array Representation cont…

Deletion

1. Start
2. Find the smallest K such that FRONT[K] <> NULL
3. Delete and Process the front element in row K of Queue
4. Stop

Insertion

1. Start
2. Insert ITEM as the rear element in row M of queue
3. Stop

Assignments

Assignment 1

Write an algorithm to convert an infix expression into it equivalent postfix expression. Explain the execution of algorithm using the following expression.

(A+B) * C – (D * E) î (F + G + (H * M))

Assignment 2

Write pseudo code to check whether a given postfix expression is correctly parenthesized

Assignment 3

Evaluate the following postfix expression:
5 3 2 * + 7 9 /4 * 2 / - 6 + 2 -

Assignment 4

Write C functions for insertion, deletion, traversal operation for circular queues

Assignment 5

Write an algorithm to copy the data elements of one stack to other without changing the order and without using any other data structure

Assignment 6

Write C functions for insertion, deletion, traversal operation for input restricted deques

Assignment 7

Write C functions for insertion, deletion, traversal operation for priority queues

Assignments

Assignment 8

Write an algorithm to check for balanced parentheses (,), {, }, [and] in an expression

Assignment 9

Write pseudo code for recursive function to display the string in reverse order.

Assignment 10

Write an algorithm to convert postfix expression to infix expression.

Assignment 11

Write an algorithm to convert prefix expression to infix expression.

Assignment 12

Write an algorithm to convert postfix expression to prefix expression.

Assignment 13

Write an algorithm to convert prefix expression to postfix expression.

Assignment 14

Write an algorithm for Tower of Hanoi (iteratively and recursively).

Assignment 15

We are given stack data structure, the task is to implement queue using only given queue data structure.


THANK
YOU!

Home Work (HW)

1. You are given 2 queues where each queue contains timestamp pair price. You have to print < price 1 price 2 > for all those timestamps where abs(ts1 – ts2) <= 1 second where ts1 and price 1 from 1st queue and ts2 and price 2 from 2nd queue.
2. Given an array, print the **Next Greater Element** (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1. Examples:

   a) For any array, rightmost element always has next greater element as -1.

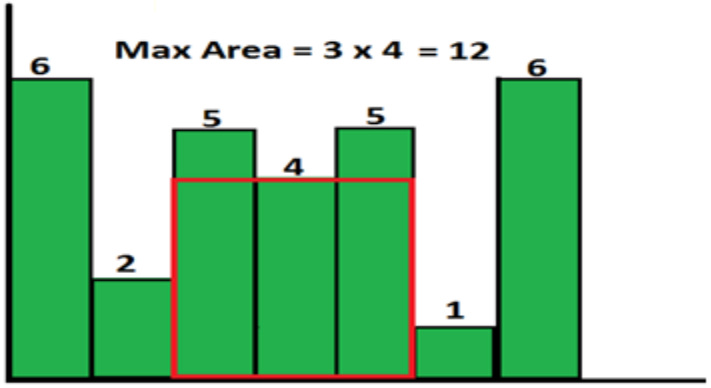
   b) For an array which is sorted in decreasing order, all elements have next greater element as -1.

   c) For the input array [4, 5, 2, 25}, the next greater elements for each element are as follows.

Element	NGE
4	5
5	25
2	25
25	-1

Home Work (HW)

3. Implement a stack using queues
4. Implement a queue using stacks
5. In a party of N people, only one person is known to everyone. Such a person may be present in the party, if yes, (s)he doesn't know anyone in the party. We can only ask questions like "does A know B? ". Find the stranger (celebrity) in minimum number of questions.
6. Find the largest rectangular area possible in a given histogram where the largest rectangle can be made of a number of contiguous bars. For simplicity, assume that all bars have same width and the width is 1 unit. For example, consider the following histogram with 7 bars of heights {6, 2, 5, 4, 5, 1, 6}. The largest possible rectangle possible is 12 (see the below figure, the max area rectangle is highlighted in red)



Home Work (HW)

7. Suppose there is a circle. There are n petrol pumps on that circle. You are given two sets of data.

1. The amount of petrol that every petrol pump has.
2. Distance from that petrol pump to the next petrol pump.

Calculate the first point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity). Expected time complexity is O(n). Assume for 1 litre petrol, the truck can go 1 unit of distance.

For example, let there be 4 petrol pumps with amount of petrol and distance to next petrol pump value pairs as {4, 6}, {6, 5}, {7, 3} and {4, 5}. The first point from where truck can make a circular tour is 2nd petrol pump. Output should be "start = 1" (index of 2nd petrol pump).

Home Work (HW)

8. Given an array of non-negative integers. Find the largest multiple of 3 that can be formed from array elements. For example, if the input array is {8, 1, 9}, the output should be "9 8 1", and if the input array is {8, 1, 7, 6, 0}, output should be "8 7 6 0".

9. Given an array and an integer k, find the maximum for each and every contiguous subarray of size k.
Examples:

Input :
arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}, k = 3
Output :
3 3 4 5 5 5 6

Input :
arr[] = {8, 5, 10, 7, 9, 4, 15, 12, 90, 13}, k = 4
Output :
10 10 10 15 15 90 90

Home Work (HW)

10. We are given queue data structure, the task is to implement stack using only given queue data structure.
11. Given a matrix of dimension m*n where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:

    0: Empty cell, 1: Cells have fresh oranges, 2: Cells have rotten oranges .

    So we have to determine what is the minimum time required so that all the oranges become rotten. A rotten orange at index [i,j] can rot other fresh orange at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1] (up, down, left and right). If it is impossible to rot every orange then simply return -1.

    Examples:
    Input: arr[][C] = { {2, 1, 0, 2, 1},
    {1, 0, 1, 2, 1},
    {1, 0, 0, 2, 1}};
    Output:
    All oranges can become rotten in 2 time frames.

Supplementary Reading

- http://www.geeksforgeeks.org/stack-data-structure/
- http://www.geeksforgeeks.org/queue-data-structure/
- https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm
- https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm
- http://www.geeksforgeeks.org/implement-stack-using-queue/
- http://www.geeksforgeeks.org/queue-using-stacks/
- http://nptel.ac.in/courses/106102064/2
- http://nptel.ac.in/courses/106102064/3
- http://freevideolectures.com/Course/2279/Data-Structures-And-Algorithms/2
- http://freevideolectures.com/Course/2279/Data-Structures-And-Algorithms/3

FAQ

What is Stack?

Stacks are data structures that allow us to insert and remove items. The operate like a stack of papers or books on our desk - we add new things to the top of the stack to make the stack bigger, and remove items from the top as well to make the stack smaller. This makes stacks a LIFO (Last In First Out) data structure – the data we have put in last is what we will get out first.

Before we consider the implementation to a data structure it is helpful to consider the operations. We then program against the specified operations. Based on the description above, we require the following functions:

```
bool stack_empty(stack S);  /* O(1), check if stack empty */
stack stack_new();          /* O(1), create new empty stack */
void push(stack S, elem e); /* O(1), add item on top of stack */
elem pop(stack S)           /* O(1), remove item from top */
```

FAQ cont…

What is Queue?

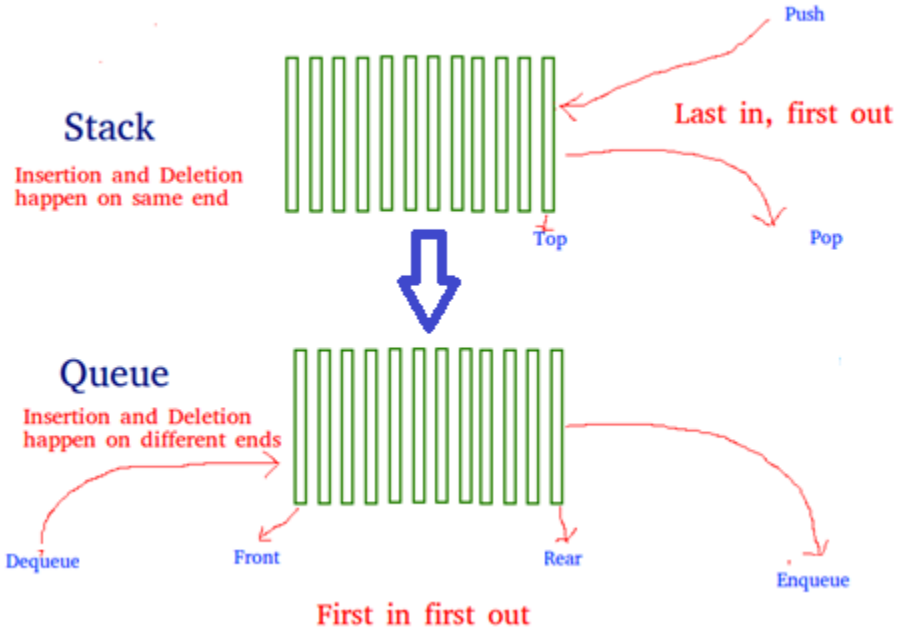
A queue is a data structure where we add elements at the back and remove elements from the front. In that way a queue is like "waiting in line": the first one to be added to the queue will be the first one to be removed from the queue. This is also called a FIFO (First In First Out) data structure.

Before we consider the implementation to a data structure it is helpful to consider the operations. We then program against the specified operations. Based on the description above, we require the following functions:

```
bool queue_empty(queue Q); /* O(1), check if queue is empty */
queue queue_new();         /* O(1), create new empty queue */
void enq(queue Q, elem s); /* O(1), add item at back */
elem deq(queue Q);         /* O(1), remove item from front */
```

FAQ cont…

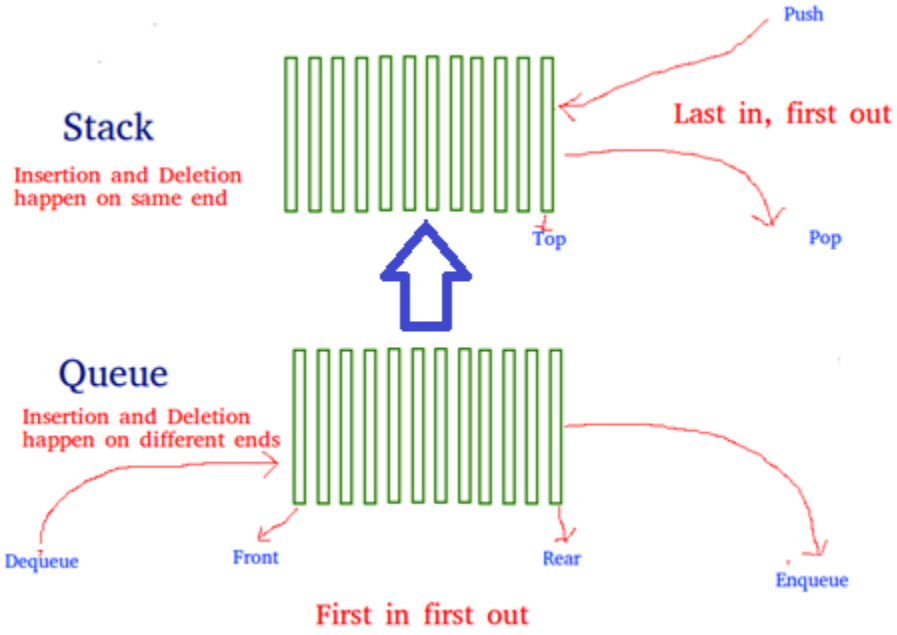
Implement Stack using Queue: We are given a stack data structure with push and pop operations, the task is to implement a queue using instances of stack data structure and operations on them.



Solution: http://www.geeksforgeeks.org/queue-using-stacks/

FAQ cont…

Implement Queue using Stack: We are given a queue data structure with enqueue and deque operations, the task is to implement a stack using instances of queue data structure and operations on them.



Solution: http://www.geeksforgeeks.org/implement-stack-using-queue/