PCCST303DATA STRUCTURES AND ALGORITHMS

PREPARED BY SHARIKA T R, ASSISTANT PROFESSOR, ASIET

Course Outcome

At the end of the course students should be able to:

Course Outcome		Bloom's Knowledge Level (KL)
CO1	Identify appropriate data structures for solving real world problems.	**K3**
CO2	Describe and implement linear data structures such as arrays, linked lists, stacks, and queues.	**K3**
CO3	Describe and Implement non linear data structures such as trees and graphs.	**K3**
CO4	Select appropriate searching and sorting algorithms to be used in specific circumstances.	**K3**

Note*: K1- Remember, K2- Understand, K3- Apply, K4- Analyse, K5- Evaluate, K6- Create*

Module 1

Basic Concepts of Data Structures Definitions; Data Abstraction; Performance Analysis - Time & Space Complexity, Asymptotic Notations;

Polynomial representation using Arrays, Sparse matrix (Tuple representation); Stacks and Queues - Stacks, Multi-Stacks, Queues, Circular Queues, Double Ended Queues; Evaluation of Expressions- Infix to Postfix, Evaluating Postfix Expressions.

Text Book

Text Books				
Sl. No	**Title of the Book**	**Name of the Author/s**	**Name of the Publisher**	**Edition and Year**
1	Fundamentals of Data Structures in C	Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed,	Universities press,	2/e, 2007
2	Introduction to Algorithms	Thomas H Cormen, Charles Leisesrson, Ronald L Rivest, Clifford Stein	PHI	3/e, 2009

System Life Cycle

The system life cycle is a series of stages that are worked through during the development of a new information system.

A lot of time and money can be wasted if a system is developed that doesn't work properly or do exactly what is required of it.

System Life Cycle-5 Phases

Requirements

Analysis

Design

Refinement & Coding

Verification

Requirements

Understanding the information you are given (the input) and what results you are to produce (the output).

We need a detailed description of the input and output which covers all cases

Analysis

Problem breakdown

- Bottom up approach
  - Start with the smallest and most specific parts of the problem and work your way up to the larger and more general parts.
  - Easier to start with small and specific problems.
  - Can be more detailed and accurate.
  - Can help to identify hidden problems.
  - Use When the problem is complex and difficult to understand.
- Top down approach
  - Start with the largest and most general parts of the problem and work your way down to the smaller and more specific parts.
  - Easier to see the big picture.
  - Can help to identify and prioritize the most important problems.
  - Can help to ensure that the individual components work together to achieve the overall goals of the system.
  - Use When the problem is well-understood and the overall goals of the system are clear.

Suppose you are designing a new website. You could use a bottom-up approach to analyze the problem by starting with the individual components of the website, such as the home page, the product pages, and the shopping cart. You could then analyze how these components interact with each other to create a user-friendly experience.

Alternatively, you could use a top-down approach to analyze the problem by starting with the overall goals of the website, such as increasing sales or generating leads. You could then analyze how the individual components of the website can be used to achieve these goals.

Design

The design phase is the process of creating a detailed plan for the software system. This plan includes the system's architecture, features, and user interface.

Steps in the Design Phase:

- Identify the needs of the system: What are the users trying to achieve with the system? What features does the system need to have?
- Design the system's architecture: How will the system's components work together? What technologies will be used?
- Design the system's features: How will the system's features be implemented? What will the user interface look like?
- Review and refine the design: Once the design is complete, it should be reviewed by users and stakeholders to ensure that it meets their needs.

Refinement and coding

Choose representations for data objects

Write algorithms for each of the operations on these objects.

Refine the algorithm

The order in which you do this may be crucial, because once you choose a representation, the resulting algorithms may be inefficient.

Verification

Develop correctness proofs for the program

Testing the program with all variety of inputs

Remove errors, if any

How to create programs

Requirements

Analysis: bottom-up vs. top-down

Design: data objects and operations

Refinement and Coding

Verification

- Program Proving
- Testing
- Debugging

Data Abstraction

Data abstraction is the process of hiding the internal details and showing only essential features of a data type.

It focuses on **what** operations are to be performed, not **how** they are implemented.

Basic Data Types in C

- char – character (1 byte)
- int – integer (2 or 4 bytes)
- float, double – real numbers
- Modifiers: short, long, unsigned

These are **built-in types** used to represent real-world data.

Grouping Data

C offers two main ways to group data:

1. **Array**
    - Collection of same type
    - Example: int list[5];
2. **Structure (struct)**
    - Collection of **different** types
    - Example:

```
struct student
{
char last_name;
int student_id;
char grade;
};
```

Pointer Types

- Every data type has a corresponding **pointer type**
- A pointer stores the **memory address** of a variable
- Example:

```
int i, *pi; // pi is a pointer to an int
```

What is a Data Type?

A data type is:
- A collection of **objects**
- A set of **operations** that can be performed on them

Example:
For int type:
- Objects: 0, +1, -1, ..., INT_MAX
- Operations: +, -, *, %, ==, !=, etc.

Why Internal Representation Matters

- char → usually 1 byte
- int → 2 or 4 bytes

- Knowing representation helps optimization

But: It can lead to issues when representation changes.

Why Hide Representation?

Hide the internal details from users.

Users access data only via **defined functions**

Allows changes in implementation without affecting the user code

Abstract Data Type (ADT)

An ADT is a data type where:

Objects & operations are clearly defined

Implementation details are hidden

This improves **modularity**, **reusability**, and **maintainability**

Example of ADT Support

- In C++: class
- In Ada: package
- In C: No built-in support, but you can **manually design ADTs**

Algorithm

Definition

An algorithm is a finite set of instructions that accomplishes a particular task.

Criteria

1. input
2. output
3. definiteness: clear and unambiguous
4. finiteness: terminate after a finite number of steps
5. effectiveness: instruction is basic enough to be carried out

Algorithm Example

You are given a set of numbers . You haveto find the largest value in thatset.

Problem Statement : Find the largestnumber in the given list of numbers?

Input : A list of positive integer numbers. (List must contain at least one number).

Output : The largest number in the givenlist of positive integer numbers.

There are many criteria for judging a program

1. Does the program meet the original specifications of the task?
2. Does it work correctly?
3. Does the program contain documentation that shows how to use it and how it woks?
4. Does the program effectively use functions to create logical units?
5. Is the code reusable?
6. Does the program efficiently use primary and secondary memory?
7. Is the programs running time acceptable for the task?

Performance Analysis

In computer science, there may be multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements.

Performance analysis helps us to select the best algorithm among the multiple algorithms designed to solve a problem.

Performance analysis of an algorithm means predicting the resources which are required to an algorithm to perform its task.

The formal definition is as follows...

"Performance analysis of an algorithm is a process of making evaluative judgment about algorithms."

Two main evaluating criteria are space and time required by that particular algorithm

Based on this, performance analysis of an algorithm can also be defined as follows

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

Complexity of an Algorithm

Performance of an algorithm is expressed as Complexity of Algorithm.

Usually the complexity of an algorithm is measured in terms of the input data size.

The complexity of an algorithm is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the input datasize 'n'.

Space required to complete the task of an algorithm is termed as Space Complexity.

Time required to complete the task of an algorithm is termed as Time Complexity of that algorithm

Mostly, the storage space required by an algorithm is simply a multiple of the data size n. Hence, generally, complexity shall refer to the running time of the algorithm

Best , Average and Worst case complexities

The complexity of an algorithm can be measured in 3 cases:

Best case complexity : - It gives the minimum possible complexity f(n).

- Example : in the case of linear search, if the item to be searched is the first element in the list itself then it can be searched with the minimum number of iteration. It will be the best case complexity of linear search.

Average case complexity :- It gives the average complexity f(n).

- In linear search the element to be searched will be in somewhat middle.

Worst case complexity : -It gives maximum possible complexity f(n) of an algorithm.

- In the case of linear search, if the element to be searched is at the last position then maximum number of iteration has to be performed.

Space complexity

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

The space requirement of an algorithm/ program depends on the following components..

- Instruction Space: It is the amount of memory used to store compiled version of instructions.
- Environmental Stack: It is the amount of memory used to store information of partially executed functions at the time of function call.
- Data Space: It is the amount of memory used to store all the variables and constants. Data space has two components:
  - Space needed for constants and simple variables in program.
  - Space needed by fixed sized structural variables, such as arrays and structures.
  - Space needed for dynamically allocated objects such as arrays and class instances.

Space Complexity $S(P)=C+S_P(I)$

Fixed Space Requirements (C)

- Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants

Variable Space Requirements ($S_P(I)$)

- depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

```
float abc (float a, float b, float c)
{
return a + b + b * c + (a + b -c) / (a + b)+ 4.00;
}
```

Ans: 3 variables , so 3*4=12 and Sabc(I)=0

Here we have float variables namely a,b and c. Assuming 8 bytes for each variable, the total space occupied by the program is 3*8 =24 bytes.

```
#include <stdio.h>
int main()
{ int n, i, sum = 0;  //4x3=12byte for 3 int
scanf("%d", &n);
 int arr[n];    //4n
 for(i = 0; i < n; i++)
 { scanf("%d", &arr[i]); sum = sum + arr[i];
}
 printf("%d", sum);
}
```

In the above-given code, the array consists of n integer elements. So, the space occupied by the array is 4 * n. Also we have integer variables such as n, i and sum. Assuming 4 bytes for each variable, the total space occupied by the program is 4n + 12 bytes.

***Program 1.11:** Recursive function for summing a list of numbers (p.20)

```
float rsum(float list[ ], int n)
{
  if (n) return rsum(list, n-1) + list[n-1];
  return 0;
}
```

$$S_{sum}(I)=S_{sum}(n)=6n$$

Assumptions:

***Figure 1.1:** Space needed for one recursive call of Program 1.11 (p.21)

Type	Name	Number of bytes
parameter: float parameter: integer return address:(used internally)	list [ ] n	2 2 2(unless a far address)
TOTAL per recursive call		6

Time complexity

Amount of time required for an algorithm to run to completion is called time complexity.

$$T(p)=\text{compile time} + \text{run time}$$

Compile time is machine dependent so we ignore compile time.

$$T(p)=tp(n)$$

To calculate execution time we will calculate step count. In step count method, we count number of times one instruction is executing.

Step count is calculated based on some basic rules,

Time Complexity

$$T(P)=C+T_P(I)$$

- Compile time (C)
  independent of instance characteristics
- run (execution) time T_P
- Definition:

  $$T_P(n)=c_aADD(n)+c_sSUB(n)+c_lLDA(n)+c_{st}STA(n)$$

  A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.
- Example
  - abc = a + b + b * c + (a + b - c) / (a + b) + 4.0
  - abc = a + b + c

Regard as the same unit
machine independent

Methods to compute the step count

Introduce variable count into programs

Tabular method

- Determine the total number of steps contributed by each statement

  step per execution x frequency

- add up the contribution of all statements

Rules that may be used for determining frequency count

Comments-0 steps

No count for { and }

Assignment statement-1 step

Return statement-1 step

Conditional statement-1 step

Loop condition for n times-n+1 steps

Body of loop=n steps

Iterative summing of a list of numbers

***Program 1.12:** Program 1.10 with count statements (p.23)

```
float sum(float list[ ], int n)
{
   float tempsum = 0; count++; /* for assignment */
   int i;
   for (i = 0; i < n; i++) {
        count++;            /*for the for loop */
        tempsum += list[i]; count++;  /* for assignment */
   }
   count++;        /* last execution of for */
   return tempsum;
   count++;        /* for return */
}
```


$\boxed{2n + 3 \text{ steps}}$

Recursive summing of a list of numbers

***Program 1.14:** Program 1.11 with count statements added (p.24)*

```
float rsum(float list[ ], int n)
{
        count++;      /*for if conditional */
        if (n) {
                count++;  /* for return and rsum invocation */
                return rsum(list, n-1) + list[n-1];
        }
        count++;
        return list[0];
}
```

$\boxed{2n+2}$

Matrix addition

***Program 1.15:** Matrix addition (p.25)*

```
void add( int a[ ] [MAX_SIZE], int b[ ] [MAX_SIZE],
                    int c [ ] [MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < rows; i++)
     for (j= 0; j < cols; j++)
       c[i][j] = a[i][j] +b[i][j];
 }
```

***Program 1.16:** Matrix addition with count statements (p.25)

```
void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],
                  int c[ ][MAX_SIZE], int row, int cols )
{
  int i, j;
  for (i = 0; i < rows; i++){
      count++; /* for i for loop */
      for (j = 0; j < cols; j++) {
        count++; /* for j for loop */
        c[i][j] = a[i][j] + b[i][j];
        count++; /* for assignment statement */
      }
      count++;    /* last time of j for loop */
  }
  count++;        /* last time of i for loop */
}
```

2rows * cols + 2 rows + 1

Tabular Method

***Figure 1.2:** Step count table for Program 1.10 (p.26)*

Iterative function to sum a list of numbers

steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Recursive Function to sum of a list of numbers

***Figure 1.3:** Step count table for recursive summing function (p.27)*

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

Matrix Addition

***Figure 1.4:** Step count table for matrix addition (p.27)

Statement	s/e	Frequency	Total steps
Void add (int a[ ][MAX_SIZE]. . .)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i = 0; i < row; i++)	1	rows+1	rows+1
for (j=0; j< cols; j++)	1	rows. (cols+1)	rows. cols+rows
c[i][j] = a[i][j] + b[i][j];	1	rows. cols	rows. cols
}	0	0	0
Total			2rows. cols+2rows+1

MATRIX MULLTIPLICATION

Statement	s/e	Frequency	Total steps
void multiply(int a[][MAX], int b[][MAX], int c[][MAX], int row1, int col1, int col2)	0	0	0
{	0	0	0
int i, j, k;	0	0	0
for (i = 0; i < row1; i++)	1	row1 + 1	row1 + 1
for (j = 0; j < col2; j++)	1	row1 × (col2 + 1)	row1 × (col2 + 1)
c[i][j] = 0;	1	row1 × col2	row1 × col2
for (k = 0; k < col1; k++)	1	row1 × col2 × (col1 + 1)	row1 × col2 × (col1 + 1)
c[i][j] += a[i][k] * b[k][j];	1	row1 × col2 × col1	row1 × col2 × col1
}	0	0	0

Total steps=row1×(2×col2+1+col2×(1+2×col1))

Let's calculate the **Time Complexity** and **Space Complexity** for a **Linear** search in case of an array.

Algorithm Linear Search (array a, n , data)	0
{	0
int i	0
for i= 1 to n	n+1
{	0
if a[i]=data	n
return i;	1
}	0
return NULL;	1
}	
Total Time	**2n + 2**

```
float sum(float list[],int n)
{
float tempsum=0;
int i;
for(i=0;i<n;i++)
        count+=2;
coumt+=3;
}
```

Frequency count

1

n+1
n

1

Ans:2n+3

A program step is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Number of steps can be calculated as a function of

- Number of inputs
- Number of outputs
- Computing time
- Magnitude of input and outputs etc

Asymptotic Notation

Complexity of an algorithm is usually a function of n.

Behavior of this function is usually expressed in terms of one or more standard functions.

Expressing the complexity function with reference to other known functions is called asymptotic complexity.

Asymptotic analysis of an algorithm, refers to defining the mathematical bound of its run-time performance.

The notations used to represent the asymptotic growth rate of algorithms are called asymptotic notations.

"Asymptotic behavior is like a race where one function (f(n)) is always chasing another (g(n)).

The first function gets closer and closer to the second, but it never quite catches up, no matter how big the input becomes."

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

Big-oh (O) Notation

The Big-oh notation determines the upper limit of f(n).

Let g(n) be the said upper limit of the function f(n)

Formal definition:

f(n) = O(g(n)) if and only if, for any two positive constants c and n0, the inequality f(n)<= c * g(n) holds for any input size n > n_0.

Example:

$f(n) = 2n + 10$

let $g(n) = n$, We can write $f(n) = O(n)$ only when the $2n + 10 <= c * n$ hold.

$2n+ 10 – c*n =0$

$(2-c)* n = -10$

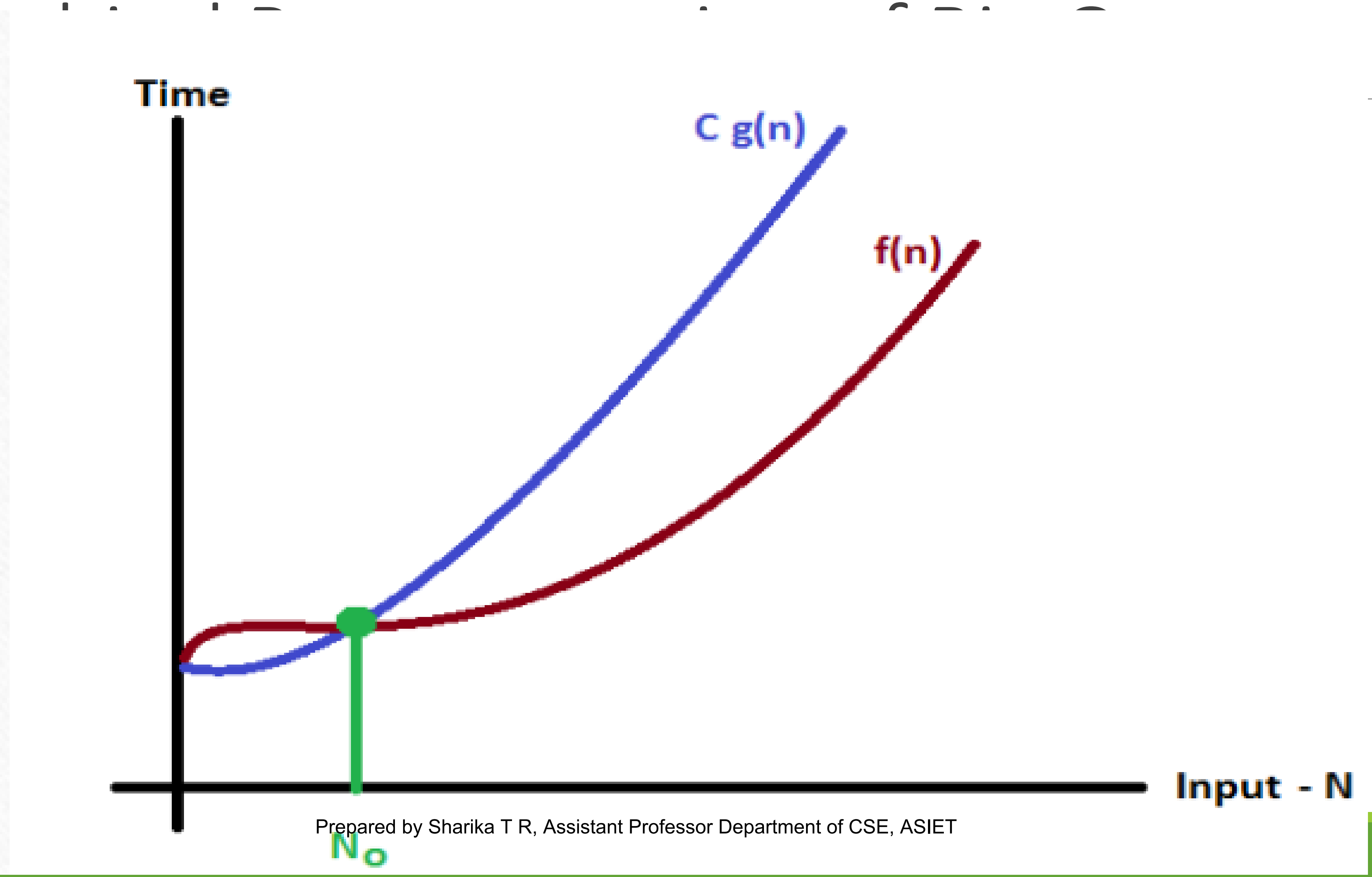
let $n =1$, $-c = -10-2 =-12 => c = 12$

This inequality hold for all the value $n_0>1$ and $c >= 12$ hence $2n + 10$ an be represented as $O(n)$

The general step wise procedure for Big-O runtime analysis is as follows:

1. Figure out what the input is and what n represents.
2. Express the maximum number of operations, the algorithm performs in terms of n.
3. Eliminate all excluding the highest order terms.
4. Remove all the constant factors.

Gra



- *Constant Multiplication:*

If $f(n) = c.g(n)$, then $O(f(n)) = O(g(n))$; where c is a nonzero constant.

- *Polynomial Function:*

If $f(n) = a_0 + a_1.n + a_2.n^2 + \text{---} + a_m.n^m$, then $O(f(n)) = O(n^m)$.

- *Summation Function:*

If $f(n) = f_1(n) + f_2(n) + \text{---} + f_m(n)$ and $f_i(n) \leq f_{i+1}(n)\ \forall\ i=1, 2, \text{---}, m$,

then $O(f(n)) = O(max(f_1(n), f_2(n), \text{---}, f_m(n)))$.

- *Logarithmic Function:*

If $f(n) = log_a n$ and $g(n)=log_b n$, then $O(f(n))=O(g(n))$

; all log functions grow in the same manner in terms of Big-O.

Running Time Complexity

- *A logarithmic algorithm – O(logn)*

Runtime grows logarithmically in proportion to n.

- *A linear algorithm – O(n)*

Runtime grows directly in proportion to n.

- *A superlinear algorithm – O(nlogn)*

Runtime grows in proportion to n.

- *A polynomial algorithm –* $O(n^c)$

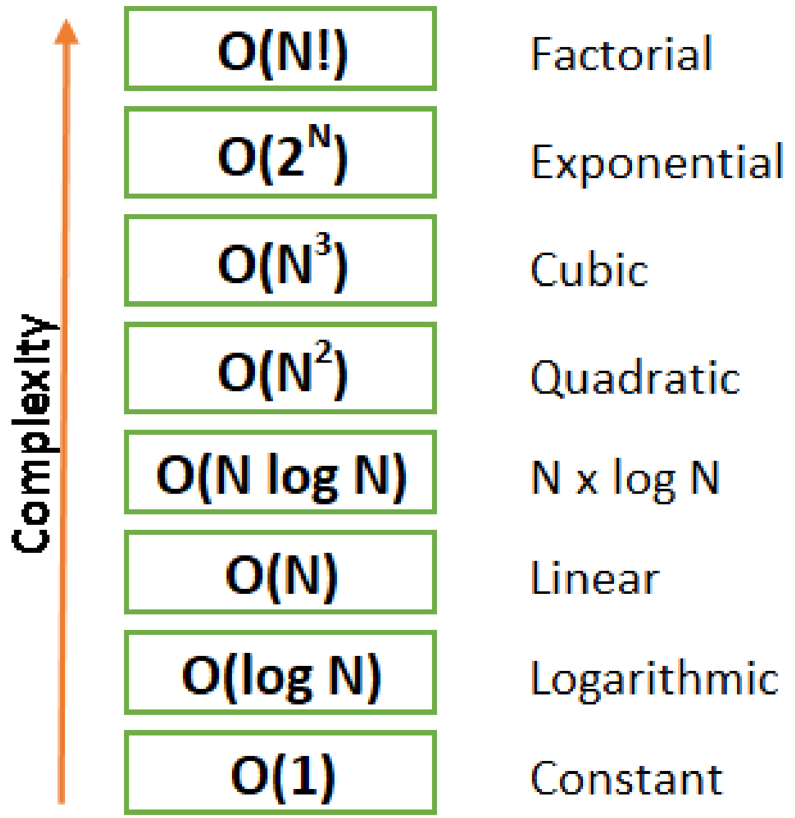
Runtime grows quicker than previous all based on n.

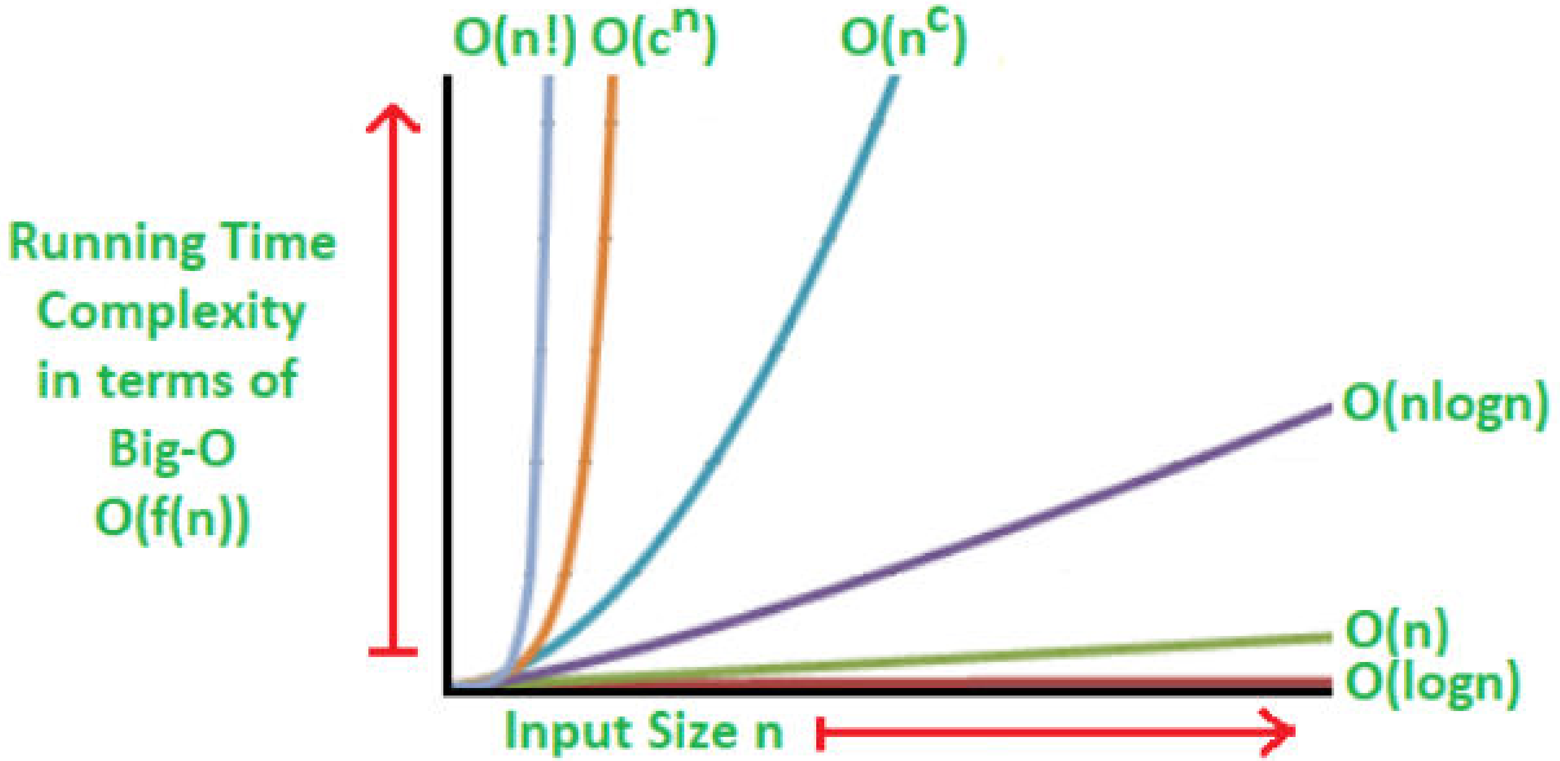
- *A exponential algorithm –* $O(c^n)$

Runtime grows even faster than polynomial algorithm based on n.

- *A factorial algorithm – O(n!)*

Runtime grows the fastest and becomes quickly unusable for even small values of n.





$O(n!), O(c^n), O(n^c)$ - Worst

$O(n\log n)$ - Bad

$O(n)$ - Fair

$O(\log n)$ - Good

$O(1)$ - Best

T(n)	Complexity
$5n^3+200n^2+15$	$O(n^3)$
$3n^2+2^{300}$	$O(n^2)$
$5\log_2 n+15 \log n$	$O(\log n)$
$2\log n^3$	$O(\log n)$
$4n+\log n$	$O(n)$
2^{64}	$O(1)$
$\text{Log } n^{10}+2\sqrt{n}$	$O(\sqrt{n})$
2^n+n^{1000}	$O(2^n)$

Big- omega(Ω) Notation

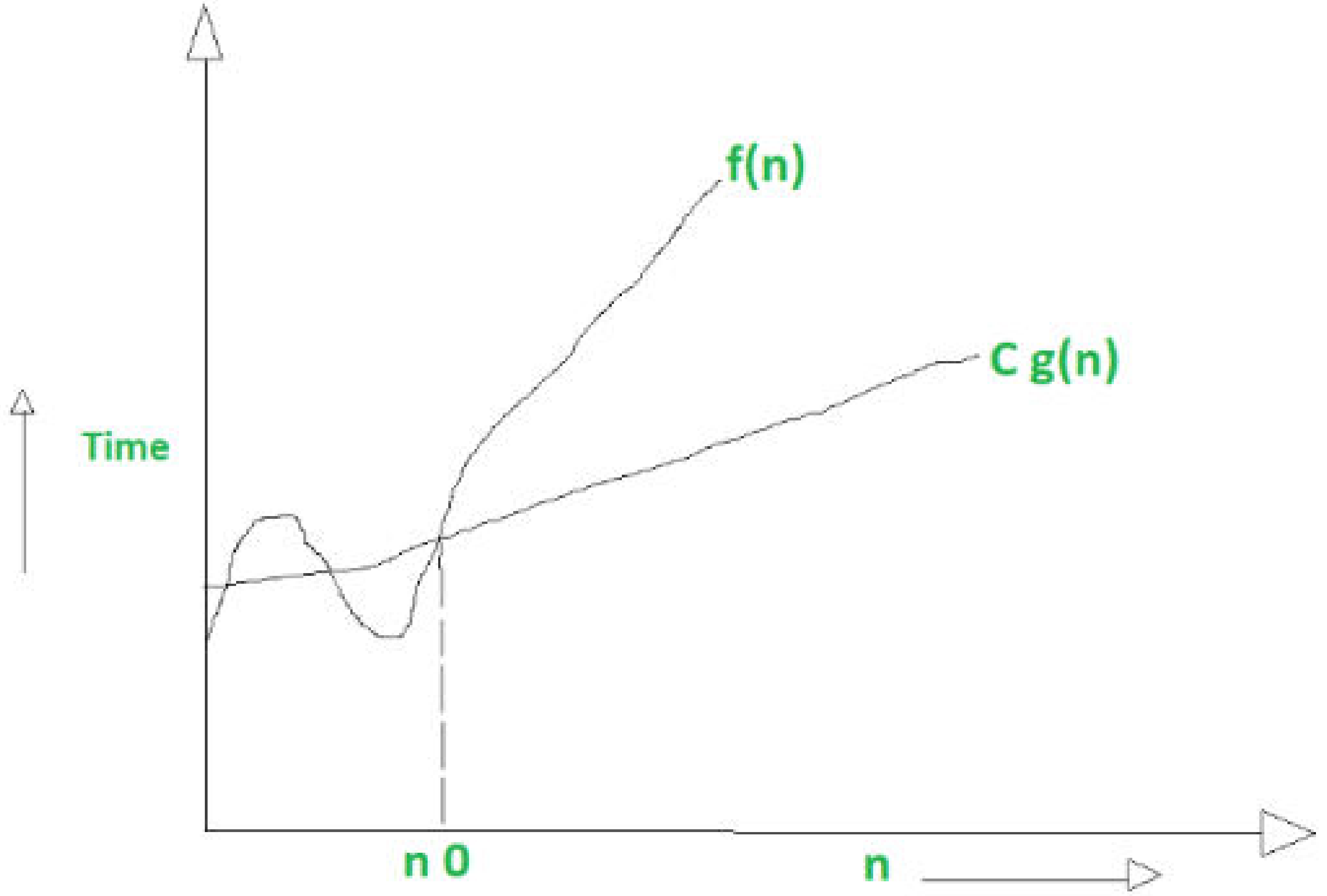
Big - Omega notation is used to define the lower bound of an algorithm in terms of Time Complexity.

Big-Omega notation always indicates the minimum time required by an algorithm for all input values.

That means Big-Omega notation describes the best case of an algorithm time complexity.

Formal Definition: $f(n) = \Omega(g(n))$ if and only if, for any two positive constants c and n0, the inequality $f(n) >= c * g(n)$ holds for any input size $n > n0$.

Explanation : Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. Then we can represent f(n) as Ω(g(n)) if and only if f(n) >= C *g(n) holds for all he input data size n >= n0 where C >=1 and n0 >= 1 are any two constants


f(n)
C g(n)
Time
n 0
n

$f(n)=2n^2+n >= g(n^2)= \Omega(n^2)$

Here g(n) should be less than or equal to f(n) it can be n^2, n, log n, loglogn, 1 etc.

But we will always take the greatest lower bound for Omega Representation

$2n^2+n >= c*n^2$

If c=2, then $2n^2 + n >= 2n^2$ ie, n>=0

Example:: Let the complexity $f(n) = 3n + 2$ and $g(n) = n$

If we want to represent $f(n)$ as $\Omega(g(n))$ then it must satisfy $f(n) >= c.g(n)$ for $C>0$ and $n_0>= 1$

$f(n) >= C\ g(n)$

$\Rightarrow 3n + 2 >= Cn$

Above condition is always TRUE for all values of $C = 1$ and $n >= 1$.

By using Big - Omega notation we can represent the time complexity as:

$3n + 2 = \Omega(n)$

$T(n) = 2n + 5$ is $W(n)$. Why?

$2n+5 >= n$, for all $n > 0$

Best case complexity

Big-theta(Ø)Notation

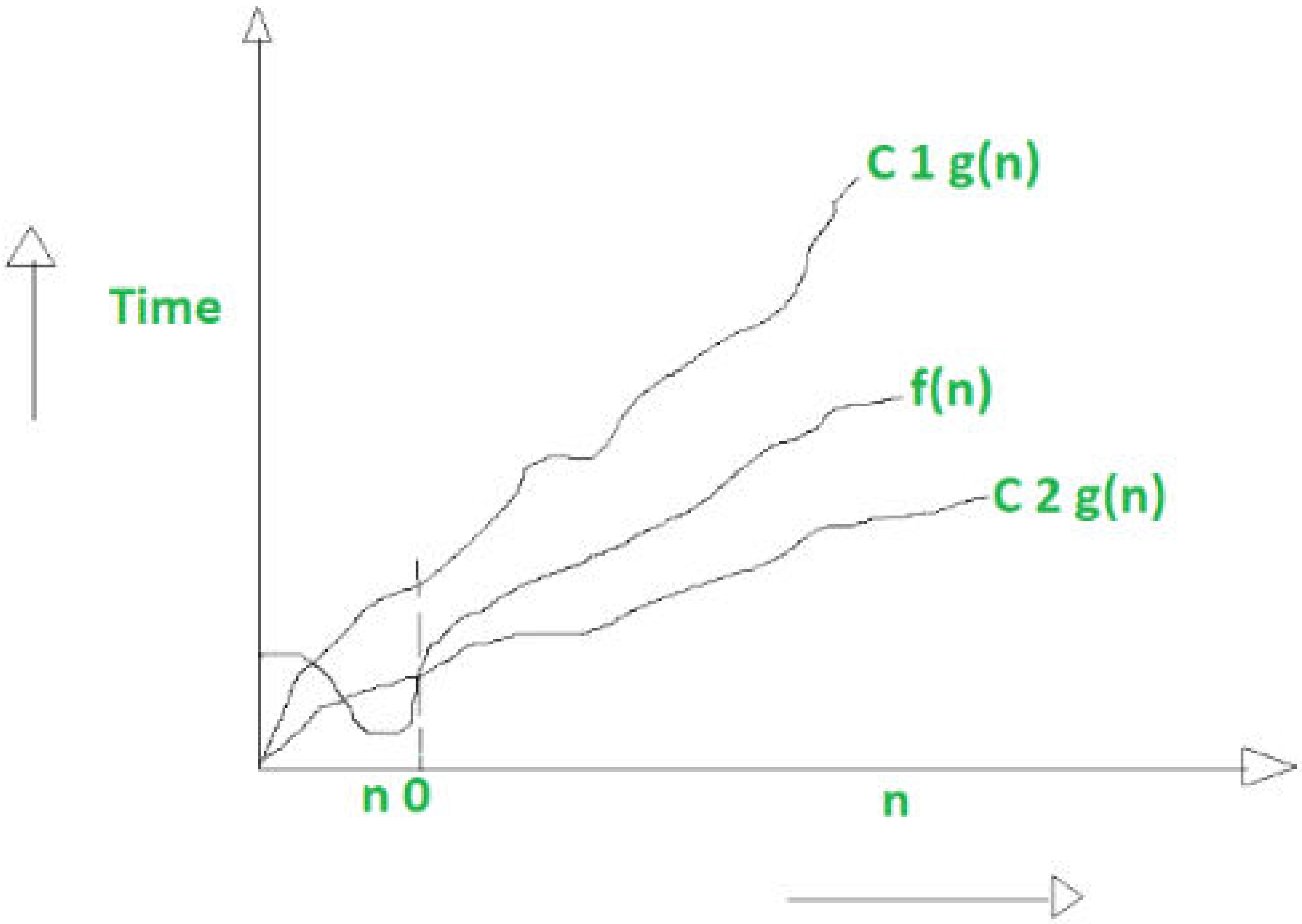
Big - Theta notation is used to define the average bound of an algorithm in terms of Time Complexity.

Big - Theta notation always indicates the average time required by an algorithm for all input values.

Big - Theta notation describes the average case of an algorithm time complexity.

Definition : $f(n) = \emptyset(g(n))$ if and only if, for any three positive constants c1,c2and n0,the inequality $c1*g(n) <= f(n) <= c2 * g(n)$ holds for any input size $n > n0$.

Explanation : - Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. Then we can represent f(n) as $\emptyset(g(n))$ if and only if $C1\ g(n) <= f(n) <= C2\ g(n)$ hold for all input data size $n >= n0$. where C1, C2 and n0 are positive constants.


C 1 g(n)
f(n)
C 2 g(n)
Time
n 0
n

Example: Let the complexity f(n) = 3n + 2 and g(n) = n

f(n) = 3n + 2 g(n) = n

If we want to represent f(n) as Ø(g(n)) then it must satisfy C1 g(n) <= f(n) <=C2 g(n) for all values of C1 > 0, C2 > 0 and n0>= 1

C1 g(n) <= f(n) <= C2 g(n) ⇒C1 n <= 3n + 2 <= C2 n

Above condition is always TRUE for all values of C1 = 1, C2 = 4 and n >= 2.

By using Big - Theta notation we can represent the time complexity as follows...

3n + 2 = Ø(n)

Example: $T(n) = 2n + 5$ is Ø(n). Why?

$2n \le 2n+5 \le 3n$, for all $n \ge 5$

Average case complexity

More examples

Big 'oh': The function $f(n)=O(g(n))$ iff there exist positive constants c and n0 such that $f(n) \leq c*g(n)$ for all n , $n \geq n0$.

Eg1:- 3n+2

$f(n)=3n+2 \leq 4n$ for all $n \geq 2$

$\leq 4*n$

$\leq O(n)$

More examples

Big 'oh': The function f(n)=O(g(n)) iff there exist positive constants c and n0 such that $f(n) \le c*g(n)$ for all n , $n \ge n0$.

Eg2:- $10n^2 + 4n + 2$

$= O(n^2)$

One possible ans

$f(n) = 10n^2 + 4n + 2 \le 11n^2$ for all $n \ge 5$

How did we find n>=5
We will check for values of n=1,2,3,4 and only after n become 5 and higher this will hold true

More examples

Omega: The function $f(n)=\Omega(g(n))$ iff there exist positive constants c and n0 such that $f(n) \geq c*g(n)$ for all n, $n \geq n0$.

Eg3:- 3n+2

One possible ans

$f(n)=3n+2 \geq 3n$ for all $n \geq 1$

$\geq 3*n$

$\geq \Omega(n)$

More examples

Omega: The function f(n)=Ω(g(n)) iff there exist positive constants c and n0 such that f(n) ≥ c*g(n) for all n, n ≥ n0 .

Eg4:- $10n^2 + 4n + 2$

f(n)= $10n^2 + 4n + 2 \geq$ n2 for all n≥1

= $\Omega(n^2)$

More examples

Theta: The function f(n)= Ø(g(n)) iff there exist positive constants c1 ,c2 and n0 such that c1 g(n) ≤ f(n) ≤ c2 g(n) for all n, n ≥ no

Eg5:- 3n+2

f(n)=3n+2

3n+2 ≤4n for all n≥2

3n+2 ≥3n for all n≥2

3n ≤ 3n+2 ≤ 4n

= Ø(n)

More examples

Theta: The function f(n)= Ø(g(n)) iff there exist positive constants c1 ,c2 and n0 such that c1 g(n) ≤ f(n) ≤ c2 g(n) for all n, n ≥ no

Eg6:- $10n^2 +4n+2$

$f(n)= 10n^2 +4n+2$

$10n^2 +4n+2 \geq n2$ for all $n \geq 5$

$10n^2 +4n+2 \leq 11n2$ for all $n \geq 5$

$n^2 \leq 10n^2 +4n+2 \leq 11n^2$

$= Ø(n^2)$

Compute time complexity of linear search algorithm using frequency count method

Algorithm	Frequency count
Algorithm LinearSearch(A[1..n], n, key)	
{	
step 1 : loc=-1	1
step 2 : repeat step 3 for i = 1 to n	n+1
step 3: if (A[i]= key)	n
step 4: loc = i	1
step 5: break	1
end if	
end repeat	
step 6: if loc == -1 then	1
step 7: print "Element not found"	
step 8: else	1
step 9: print " Element found at " loc	
step10: stop	
}	
	2n +6

What is the purpose of calculating frequency count? Compute the frequency count of the following code fragment.

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        printf("%d",a[i][j]);
```

Derive the Big O notation for $f(n) = n^2+2n+5$.

$f(n) = n^2+2n+5$.

let $g(n) = n^2$, $f(n) = Og(n)$ iff.

$f(n) <= c*g(n)$

$n^2+2n+5 <= c* n^2$ $n^2+2n+5 - c* n^2 =0$

→

$(1-c) n^2+2n+5=0$

let $n =1$

$(1-c)+2+5 =0$

$-c +8 =0 => c = 8$.

$n^2+2n+5 = O(n^2)$, for all the values of $c >=8$ and $n0 >=1$

$N^2 + N = O(N^3)$ Justify your answer..

$N^2 + N = O(N^3)$ iff

$N^2 + N <= c*N^3$

$N^2 + N - c*N^3 = 0$

let $N = 1$

$1+1-c = 0$

$c = 2$

Hence we can say that $N^2 + N = O(N^3)$ for all $c >= 2$ and

$N0 >= 1$

What is a polynomial

A polynomial p(x) is the expression in variable x which is in the form

$$ax^n + bx^{n-1} + ... + jx + k$$

where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

1. one is the coefficient
2. other is the exponent

Example

$$10x^2 + 26x,$$

here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

The sign of each coefficient and exponent is stored within the coefficient and the exponent itself

Additional terms having equal exponent is possible one

The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent

Array Representation of Polynomial

Arrays can be used to represent and manipulate polynomials in a single variable.

For eg:

$A(x)=3x^2+2x-4$

$B(x)=x^8-10x^5-3x^3+1$

The polynomial A(x) has 3 terms → $3x^2$, $2x$ and -4

The coefficients of A(x) are → 3, 2 and -4

The exponents are → 2, 1 and 0

A term of a polynomial can be represented as a (coefficient, exponent) pair

For eg: (3,2) represents $3x^2$

A term whose coefficient is non-zero is called a nonzero term

The degree of a polynomial is the largest exponent from among the nonzero terms

Arrange the terms in decreasing order of exponent. This simplifies many of the operations

Use a 2D array to represent coefficient and exponent pair of a polynomial

Eg: $A(x)=3x^2+2x-4$

3	2	-4
2	1	0

Eg: $A(x)=5x^5+3x^4-2x^3+9$

5	3	-2	9
5	4	3	0

Add $A(x)=3x^2+2x-4$ and $B(x) = 5x^5+3x^4-2x^2+9$

3	2	-4
2	1	0

+

5	3	-2	9
5	4	2	0

5	3	1	2	5
5	4	2	1	0

Another example denoted in matrix form

A(X)

3	2
2	1
-4	0

B(X)

3	4
3	3
-2	2
9	0

C(X)

3	4
3	3
1	2
2	1
5	0

Algorithm POLY_ADDITION

Input: Two polynomial POLY1 and POLY2 with size of the row ptr1 and ptr2.

Output: Sum of two polynomial RESULT with row size, rptr

Data Structure: Polynomial is implemented using array.

Steps:

i=0,j=0, rptr=0

ptr1=row size of(POLY1), ptr2= row size of(POLY2)

While(i<ptr1 AND j<ptr2)

    if(POLY1[i][1]=POLY2[j]][1]) then

        RESULT[rptr][0]=POLY1[i][0]+POLY2[j][0]

        RESULT[rptr][1]=POLY1[i][1]

        i=i+1,

        j=j+1,

        rptr=rptr+1

```
ElseIf( POLY1[i][1]>POLY2[j][1])
   RESULT[rptr][0]=POLY1[i][0]
   RESULT[rptr][1]=POLY1[i][1]
   i=i+1,rptr=rptr+1
Else
   RESULT[rptr][0]=POLY2[j][0]
   RESULT[rptr][1]=POLY2[j][1]
j=j+1,rptr=rptr+1
EndIf
EndWhile
```

```
While(i<ptr1)
   RESULT[rptr][0]=POLY1[i][0]
   RESULT[rptr][1]=POLY1[i][1]
   i=i+1,rptr=rptr+1
EndWhile
While(j<ptr2)
   RESULT[rptr][0]=POLY2[j][0]
   RESULT[rptr][1]=POLY2[j][1]
   j=j+1,rptr=rptr+1
EndWhile
Stop
```

```
#include <stdio.h>

#define MAX 100

int main() {
    int POLY1[MAX][2], POLY2[MAX][2], RESULT[MAX][2];
    int ptr1, ptr2, rptr = 0;
    int i = 0, j = 0;

    // Input size of polynomials
    printf("Enter the number of terms in POLY1: ");
    scanf("%d", &ptr1);
    printf("Enter terms of POLY1 (coefficient and exponent):\n");
    for (i = 0; i < ptr1; i++) {
        scanf("%d %d", &POLY1[i][0], &POLY1[i][1]);
    }

    printf("Enter the number of terms in POLY2: ");
    scanf("%d", &ptr2);
    printf("Enter terms of POLY2 (coefficient and exponent):\n");
    for (j = 0; j < ptr2; j++) {
        scanf("%d %d", &POLY2[j][0], &POLY2[j][1]);
    }
```

```
// Reset i, j for algorithm
  i = 0;
  j = 0;

  // Polynomial addition
  while (i < ptr1 && j < ptr2) {
    if (POLY1[i][1] == POLY2[j][1]) {
      RESULT[rptr][0] = POLY1[i][0] + POLY2[j][0];
      RESULT[rptr][1] = POLY1[i][1];
      i++;
      j++;
      rptr++;
    } else if (POLY1[i][1] > POLY2[j][1]) {
      RESULT[rptr][0] = POLY1[i][0];
      RESULT[rptr][1] = POLY1[i][1];
      i++;
      rptr++;
    } else {
      RESULT[rptr][0] = POLY2[j][0];
      RESULT[rptr][1] = POLY2[j][1];
      j++;
      rptr++;
    }
  }
```

```
// Copy remaining terms of POLY1
while (i < ptr1) {
    RESULT[rptr][0] = POLY1[i][0];
    RESULT[rptr][1] = POLY1[i][1];
    i++;
    rptr++;
}
// Copy remaining terms of POLY2
while (j < ptr2) {
    RESULT[rptr][0] = POLY2[j][0];
    RESULT[rptr][1] = POLY2[j][1];
    j++;
    rptr++;
}
// Display result
printf("Sum of the two polynomials is:\n");
for (i = 0; i < rptr; i++) {
    printf("%dx^%d", RESULT[i][0], RESULT[i][1]);
    if (i != rptr - 1)
        printf(" + ");
}
printf("\n");
return 0;
}
```

SPARSE MATRIX

It is a special array that contains more number of zero values than the non-zero values for their elements

Eg:

No of zero elements =6

No. of non zero elements = 3

Therefore, it’s a sparse matrix

0	1	7
0	0	0
0	0	2

SPARSE MATRIX

A sparse matrix =2D sparse array

A matrix is said to be a sparse matrix if most of its elements are zero.

dense matrix =A matrix that is not sparse

The density of a matrix is the percentage of entries that are non-zero

Alternative Representations

If most of the elements are zero then the occurrence of zero elements in a large array is both a computational and storage inconvenience

Alternative Representations

1. Array representation
2. Dynamic representation

ARRAY Representation (Tuple matrix)

All non-zero elements are stored in another array of triplet

- No of rows in the new array = No. of non-zero elements + 1
- No. of columns in the new array = 3

Triplet contains

1. row number of the non-zero element
2. column number of the non-zero element
3. Value of non-zero element

Triplet can be represented by

- <Row, Col, Element>

ARRAY Representation

Example: Sparse Matrix

No of zero elements =10

No. of non zero elements = 2

Therefore, it's a sparse matrix

1	0	0	0
0	0	0	5
0	0	0	0

Tuple matrix

(0,0) – No of rows in sparse matrix

(0,1) – No of columns in sparse matrix

(0,2) – No of non-zero elements in sparse matrix

3	4	2
0	0	1
1	3	5

0	0	0
0	0	0
6	0	0
0	0	2

4	3	2
2	0	6
3	2	2

Convert sparse matrix to tuple matrix

0	0	0	3
1	0	0	0
2	0	0	0

3	4	3
0	3	3
1	0	1
2	0	2

Sparse Matrix to tuple matrix

k=1, i=0,j=0

0	0	0
0	0	0
6	0	0
0	0	2

4	3	2
2	0	6
3	2	2

1. While(i<r)
    1. While(j<c)
        1. If (A[i][j]!=0)
            1. TUPLE[k][0]=i
            2. TUPLE[k][1]=j
            3. TUPLE[k][2]=A[i][j]
            4. k=k+1, count = count +1
        2. EndIf
    2. j=j+1
    3. EndWhile
2. i=i+1
3. EndWhile

4. TUPLE[0][0]=r
5. TUPLE[0][1]=c
6. TUPLE[0][2]=count

Algorithm :

SparseMatrix_to_TupleMatrix

Input: A sparse Matrix A

Output: Matrix in tuple form

Data Structure: A Matrix with r-rows and c-columns

Steps:

1. i=0, j=0, k=1,count=0,
2. While(i<r)
   1. While(j<c)
      1. If (A[i][j]!=0)
         1. TUPLE[k][0]=i
         2. TUPLE[k][1]=j
         3. TUPLE[k][2]=A[i][j]
         4. k=k+1, count = count +1
      2. EndIf
   2. j=j+1
   3. EndWhile
3. i=i+1
4. EndWhile
5. TUPLE[0][0]=r
6. TUPLE[0][1]=c
7. TUPLE[0][2]=count
8. end

Sparse Matrix addition

0	4	0	0
0	0	7	0
6	0	0	0

0	0	0	3
1	0	0	0
2	0	0	0

Convert
sparse matrix
to
tuple matrix

3	4	3
0	1	4
1	2	7
2	0	6

3	4	3
0	3	3
1	0	1
2	0	2

3	4	3
0	1	4
1	2	7
2	0	6

+

3	4	3
0	3	3
1	0	1
2	0	2

→

3	4	5
0	1	4
0	3	3
1	0	1
1	2	7
2	0	8

Sparse Matrix addition – case 1

3	4	3
0	1	4
1	2	7
2	0	6

+

3	4	3
1	0	3
1	2	1
2	0	2

→

3	4	4
0	1	4
1	0	3
1	2	8
2	0	8

```
If ((TUPLE1 [i][0] < TUPLE2 [j][0] ))
        SUM [ptr][0] =TUPLE1 [i][0]
        SUM [ptr][1] =TUPLE1 [ i][1]
        SUM [ptr][2] =TUPLE1 [ i][2]
        i=i+1,
        ptr=ptr+1,
        elem=elem+1
```

Sparse Matrix addition – case 2

3	4	3
1	0	4
1	2	7
2	0	6

+

3	4	3
0	1	3
1	2	1
2	0	2

→

3	4	4
0	1	3
1	0	4
1	2	8
2	0	8

```
If ((TUPLE1 [i][0] > TUPLE2 [j][0] ))
        SUM [ptr][0] =TUPLE2 [j][0]
        SUM [ptr][1] =TUPLE2 [ j][1]
        SUM [ptr][2] =TUPLE2 [ j][2]
        j=j+1,
        ptr=ptr+1,
        elem=elem+1
```

Sparse Matrix addition – case 3

3	4	3
0	1	4
1	2	7
2	0	6

+

3	4	3
0	1	3
1	0	1
2	0	2

3	4	4
0	1	7
1	0	1
1	2	7
2	0	8

If((TUPLE1 [i][0] = TUPLE2 [j][0]) AND (TUPLE1 [i][1] =TUPLE2 [j][1]))

SUM [ptr][0] =TUPLE1 [i][0]
SUM [ptr][1] =TUPLE1 [i][1]
SUM [ptr][2] =TUPLE1 [i][2] +TUPLE2 [j][2]
Ptr=ptr+1,
i=i+1,
j=j+1,
elem=elem+1

Sparse Matrix addition – case 4

3	4	3
0	1	4
1	0	7
2	0	6

+

3	4	3
0	1	3
1	2	1
2	0	2

→

3	4	4
0	1	7
1	0	7
1	2	1
2	0	8

```
If ((TUPLE1 [i][0] = TUPLE2 [j][0] ) && (TUPLE1 [i][1] <TUPLE2 [j][1] ) )
        SUM [ptr][0] =TUPLE1 [i][0]
        SUM [ptr][1] =TUPLE1 [ i][1]
        SUM [ptr][2] =TUPLE1 [ i][2]
        i=i+1,
        ptr=ptr+1,
        elem=elem+1
```

Sparse Matrix addition – case 5

3	4	3
0	1	4
1	2	7
2	0	6

+

3	4	3
0	1	3
1	0	1
2	0	2

3	4	4
0	1	7
1	0	1
1	2	7
2	0	8

```
If ((TUPLE1 [i][0] = TUPLE2 [j][0] ) && (TUPLE1 [i][1] >TUPLE2 [j][1] ) )
        SUM [ptr][0] =TUPLE2 [j][0]
        SUM [ptr][1] =TUPLE2[ j][1]
        SUM [ptr][2] =TUPLE2[ j][2]
        j=j+1,
        ptr=ptr+1,
        elem=elem+1
```

Visit sharikatr.in for more notes and ppts

3	4	3
1	0	4
1	1	7
1	2	6

\+

3	4	3
0	0	3
0	1	1
0	2	2

→

3	4	6
0	0	3
0	1	1
0	2	2
1	0	4
1	1	7
1	2	6

```
While(i<n1)
    SUM[ptr][0]=TUPLE1[i][0]
    SUM[ptr][1]=TUPLE1[i][1]
    SUM[ptr][2]=TUPLE1[i][2]
    i=i+1,
    ptr=ptr+1,
    elem=elem+1
EndWhile
```

Sparse Matrix addition – case 6a

Sparse Matrix addition – case 6b

3	4	3
0	0	4
0	1	7
0	2	6

+

3	4	3
1	0	3
1	1	1
1	2	2

→

3	4	6
0	0	4
0	1	7
0	2	6
1	0	3
1	1	1
1	2	2

```
While(j<n2)
    SUM[ptr][0]=TUPLE2[j][0]
    SUM[ptr][1]=TUPLE2[j][1]
    SUM[ptr][2]=TUPLE2[j][2]
    j=j+1,
    ptr=ptr+1,
    elem=elem+1
EndWhile
```

Stack

Linear Data structure

Ordered collection of homogeneous data elements where insertion and deletion takes place at one end only.

Eg: shunting of trains in a railway yard, Plates on a tray, Stack of books

Last In First Out order

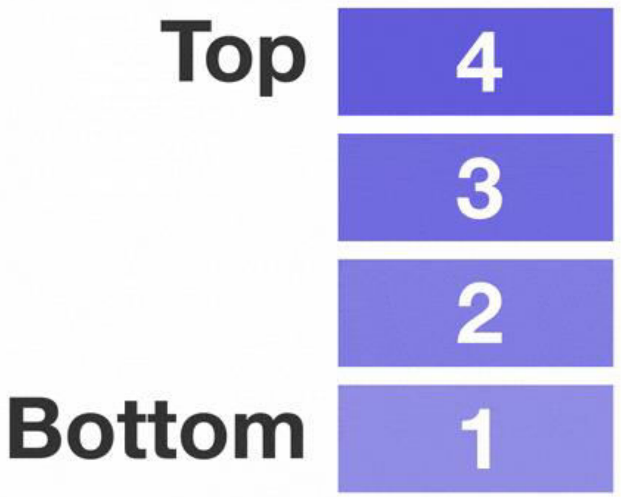
Static implementation -using array. It is a very simple technique, but it is not flexible. The size of the stack has to be declared during program design and after that size cannot be varied

Dynamic implementation –using Linked List. It uses pointers to implement stack. It is more efficient

- TOP>=SIZE-1 – overflow, stack is full
- Top= -1 –underflow, stack is empty

Operations

1. PUSH
2. POP



Insertion and deletion is at TOP of the stack

An element in a stack : ITEM

The maximum no of elements a stack can accommodate : SIZE

PUSH

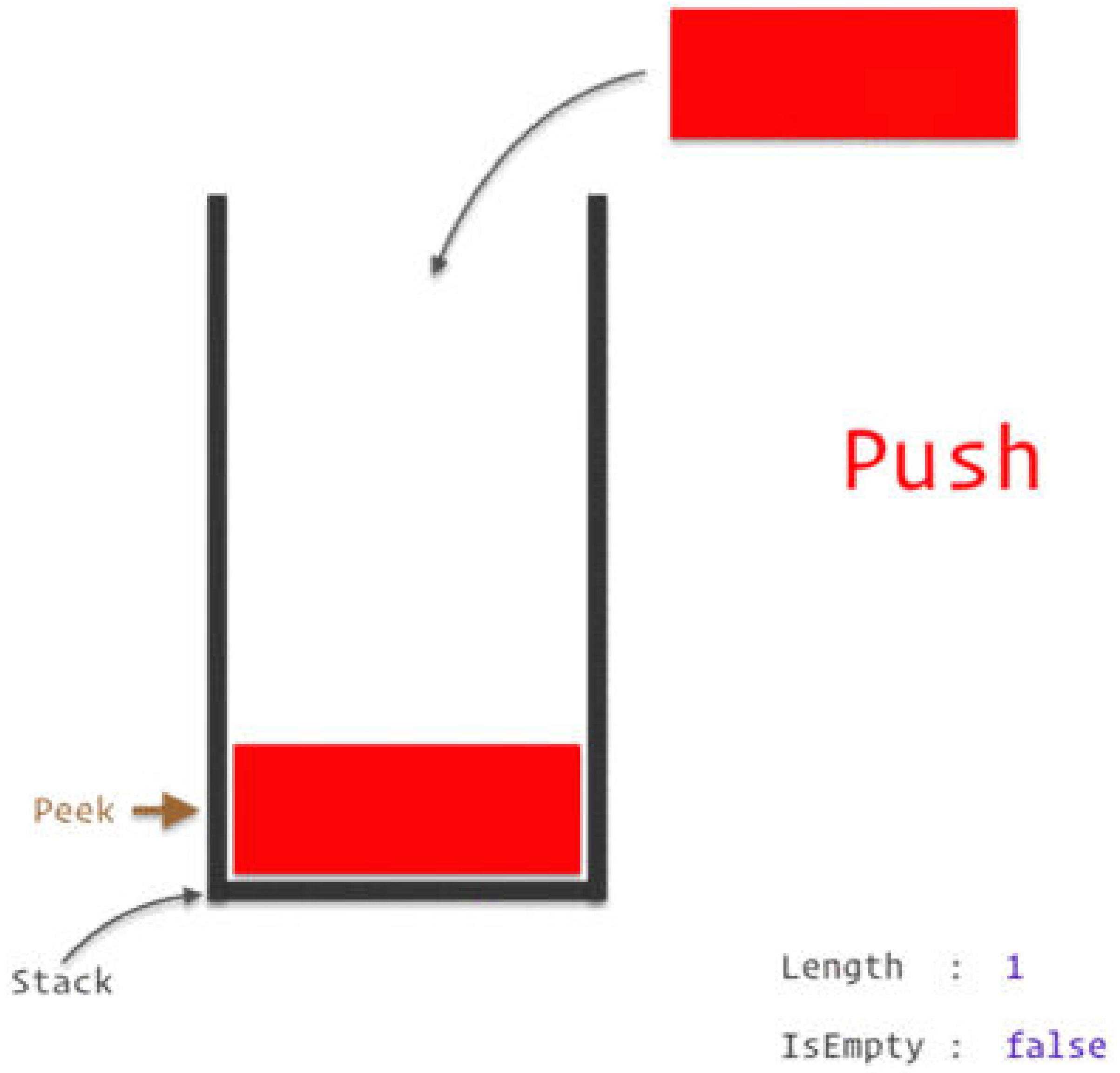
Insert an item to stack

Algorithm Push_Array

- Input: The new item ITEM to be pushed.
- Output: A stack with newly pushed ITEM at the TOP position
- DS: An array A with TOP as the pointer.

Steps:

1. If TOP>=SIZE-1 then
   i. Print " Stack is full"
2. Else
   i. TOP=TOP+1
   ii. A[TOP]=ITEM
3. EndIf
4. Stop.


Push
Peek
Stack
Length : 1
IsEmpty : false

POP

Delete an item to stack

Algorithm Pop_Array

- Input: A stack with elements.
- Output: Remove an ITEM from the TOP of stack if it is not empty
- DS: An array A with TOP as the pointer.

Steps:

1. If TOP < 0 then
   i. Print " Stack is empty"
2. Else
   i. ITEM=A[TOP]
   ii. TOP=TOP-1
3. EndIf
4. Stop.

Stack -Status

- Algorithm Status_Array
- Input: A stack with elements.
- Output: State whether it is empty or full, available free space and item at TOP
- DS: An array A with TOP as the pointer.

Steps:

1. If TOP<0 then
   i. Print " Stack is empty"
2. Else
   i. If TOP>=SIZE-1 then
      a) Print " Stack is full"
   ii. Else
      a) Print "Element at the TOP is", A[TOP]
      b) Free=(SIZE-1-TOP)/SIZE*100
      c) Print "Percentage of free stack is",free
   iii. Endif
3. EndIf
4. Stop.

Applications of stack

1. String Reversal
2. Evaluation of Arithmetic expression
    1. Infix to postfix conversion
    2. Postfix evaluation
3. Activation Record Management
4. Multiple Stack
5. Tower Hanoi

Evaluation of Arithmetic expression

Three types of notation:

1. Infix notation: operator is written between the operands (A+B)

2. Prefix notation: operator is written before the operands, also called polish notation (+AB)

3. Postfix notation: operator is written after operands, also known as suffix or reverse Polish notation (AB+)

Advantages of using postfix notation:

Human beings are quite used to work with infix notation, but infix is much complex and required to remember set of rules. (E.g. precedence and associativity)

Computer have to scan left to right several times to evaluate infix expression

Postfix is much easy to work, and no need for operator precedence and other rules.

Computer can evaluate an expression in a single scan

Infix to Postfix

Rules to be remembered during infix to postfix conversion:

1. Parenthesize the expression starting from left to right.
2. During parenthesize the expression, operands associated with operator having higher Precedence are first parenthesized.
3. The sub expression which has been converted into postfix is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Order of Precedence (highest to lowest)

Exponentiation

Multiplication/division *, /

Addition/subtraction +, -

For operators of same precedence, the left-to-right rule applies:

A+B+C means (A+B)+C.

Operators	Symbols
Parenthesis	( ), {}, [ ]
Exponents	^
Multiplication and Division	*, /
Addition and Subtraction	+ , -

Infix	Postfix
A + B	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *
A B * C – D + E/F	A B C*D – E F/+

You should formulate the conversion algorithm using the following six rules:

1. Scan the input string (infix notation) from left to right. One pass is sufficient.
2. If the next symbol scanned is an operand, it may be immediately appended to the postfix string.
3. If the next symbol is an operator,
   i. Pop and append to the postfix string every operator on the stack that
      a) is above the most recently scanned left parenthesis, and
      b) has precedence higher than or is a right-associative operator of equal precedence to that of the new operator symbol.
   ii. Push the new operator onto the stack.

4. When a left parenthesis is seen, it must be pushed onto the stack.
5. When a right parenthesis is seen, all operators down to the most recently scanned left parenthesis must be popped and appended to the postfix string. Furthermore, this pair of parentheses must be discarded.
6. When the infix string is completely scanned, the stack may still contain some operators. [Why are there no parentheses on the stack at this point?] All the remaining operators should be popped and appended to the postfix string.

ALGORITHM

1. Scan all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is an operand, then immediately append it to the Postfix Expression.
3. If the reading symbol is left parenthesis '(', then Push it onto the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of the stack until the respective left parenthesis is popped and append each popped symbol to Postfix Expression.
5. If the reading symbol is an operator (+, –, *, /), then Push it onto the Stack. However, first, pop the operators which are already on the stack that have higher or equal precedence than the current operator and append them to the postfix. If an open parenthesis is there on top of the stack then push the operator into the stack.
6. If the input is over, pop all the remaining symbols from the stack and append them to the postfix.

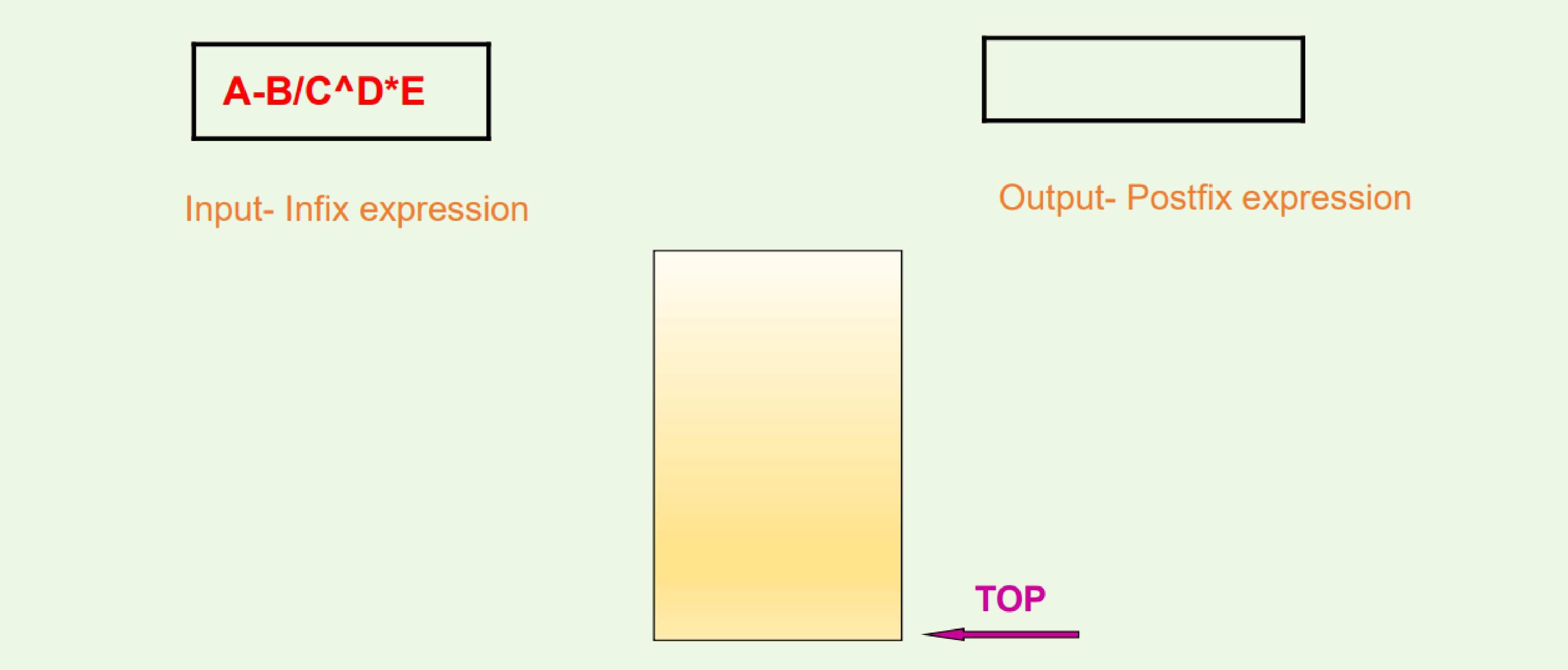
infix to postfix

Implementation :

A-B/C^D*E

Input- Infix expression

Output- Postfix expression

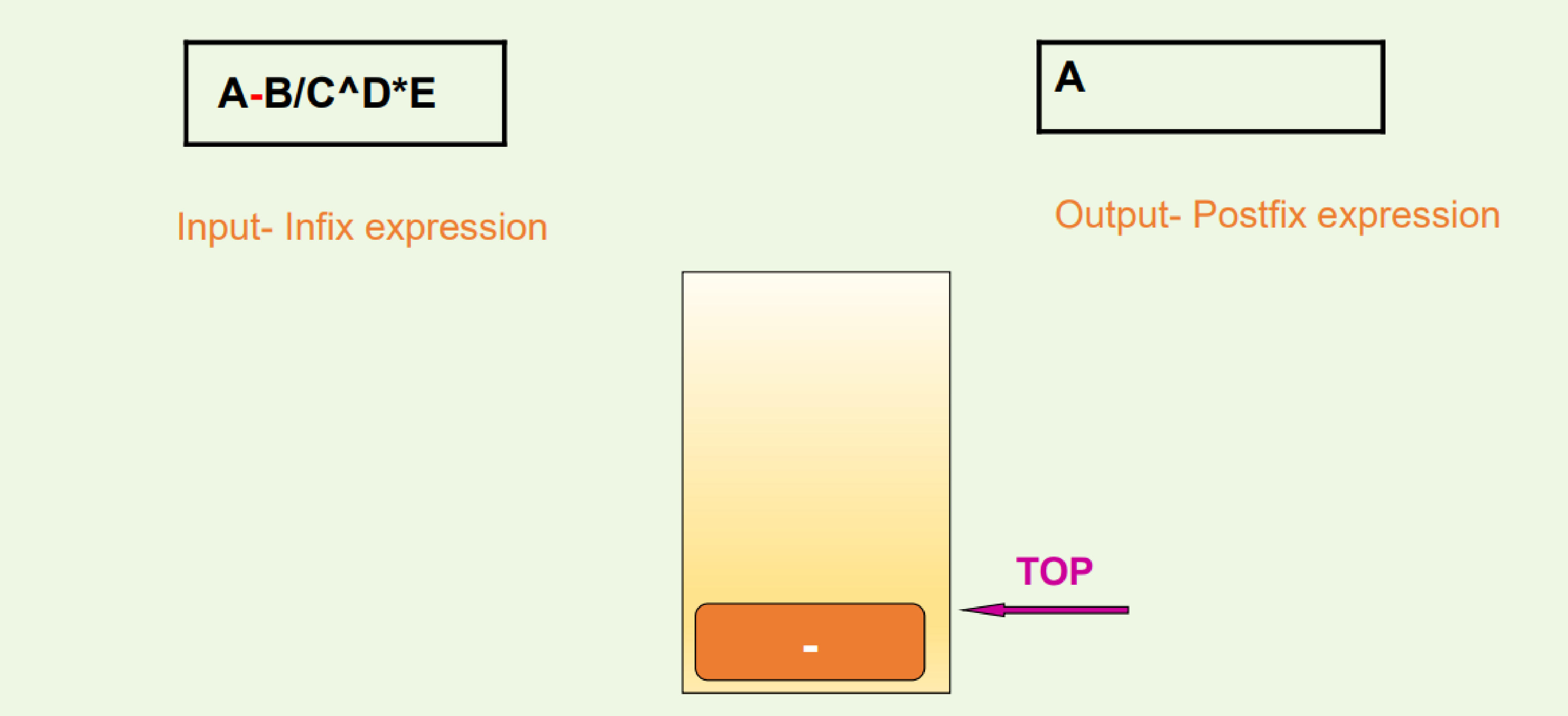

A-B/C^D*E
Input- Infix expression
Output- Postfix expression
TOP

A-B/C^D*E

A

Input- Infix expression

Output- Postfix expression

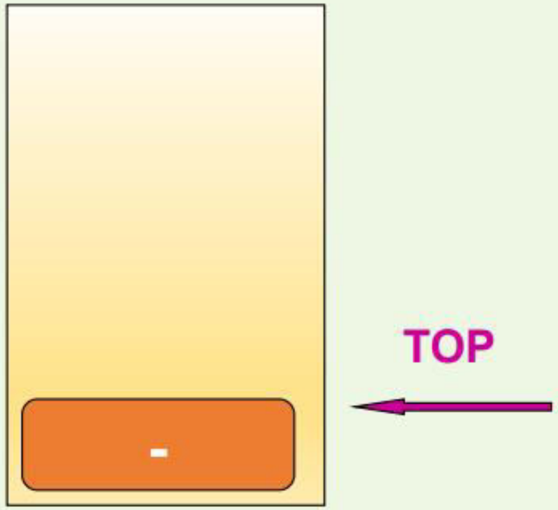

A-B/C^D*E
Input- Infix expression
A
Output- Postfix expression
TOP
-

A-B/C^D*E

AB

Input- Infix expression

Output- Postfix expressio

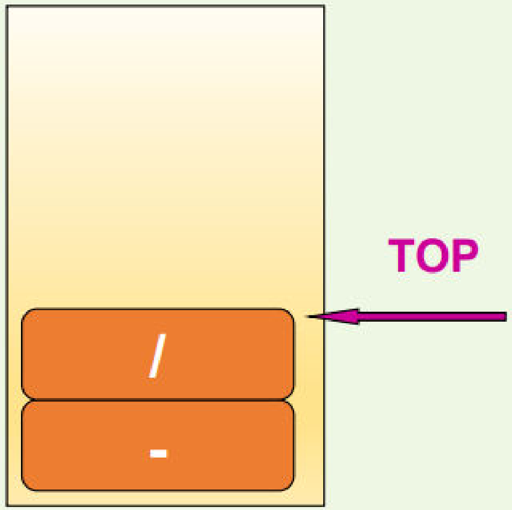


A-B / C^D*E

Input- Infix expression

AB

Output- Postfix expression



Topstack has less precedence so push to stack

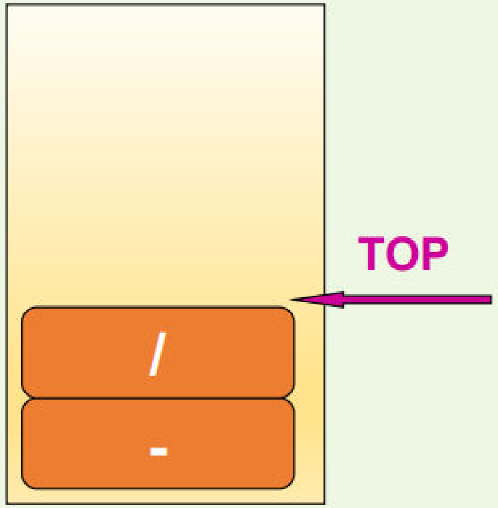
Implementation :

A-B / C^D*E

Input- Infix expression

ABC

Output- Postfix expression

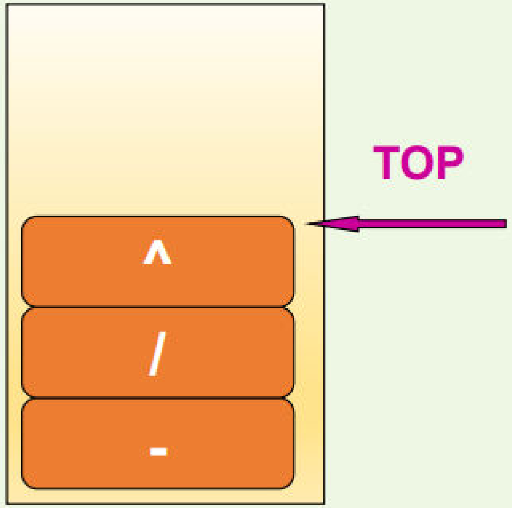


A-B / C^D*E

Input- Infix expression

ABC

Output- Postfix expression



Topstack has less precedence so push to stack

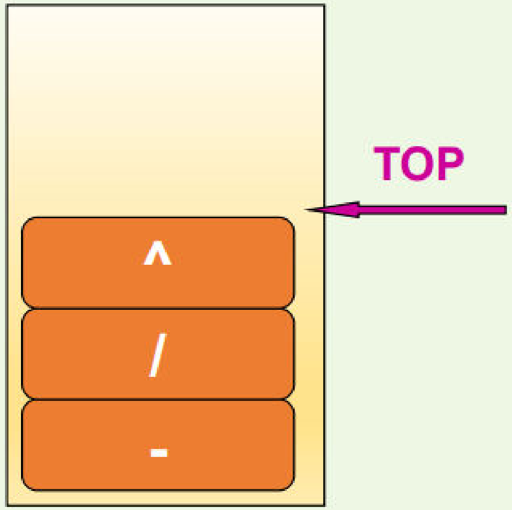
Implementation :

A-B / C^D*E

ABCD

Input- Infix expression

Output- Postfix expression

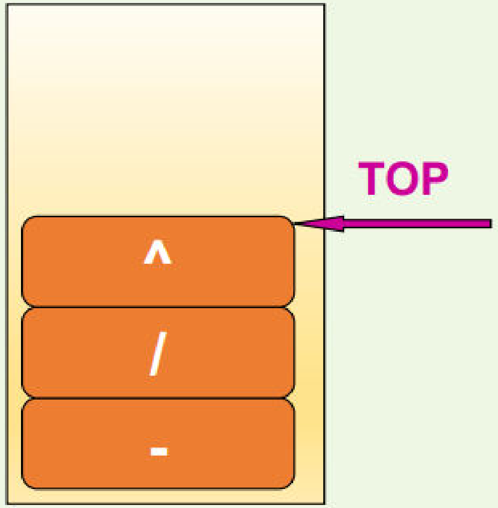


A-B / C^D*E

Input- Infix expression

ABCD

Output- Postfix expression



Topstack has more precedence so POP stack

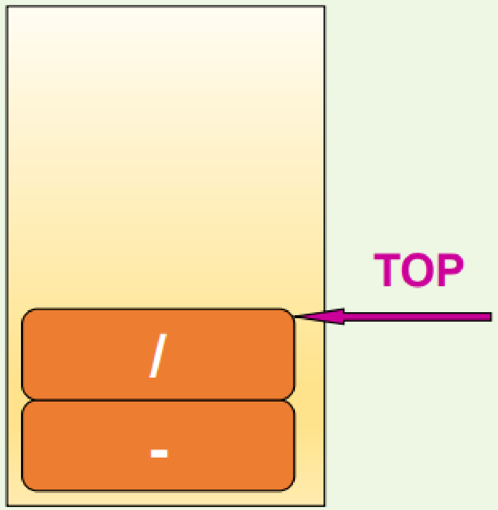
Implementation :

A-B / C^D*E

ABCD^

Input- Infix expression

Output- Postfix expression



Topstack has equal precedence so POP stack

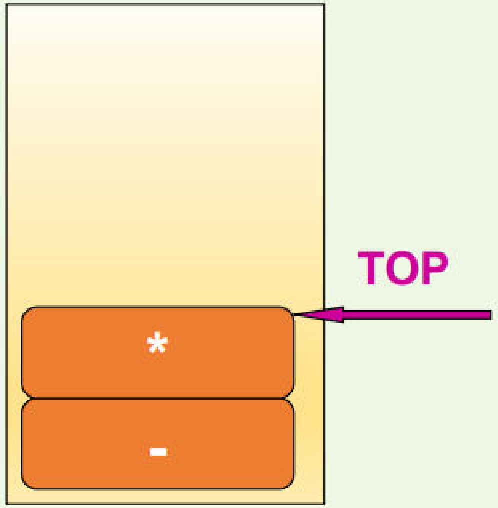
Implementation :

A-B / C^D*E

Input- Infix expression

ABCD^/

Output- Postfix expression



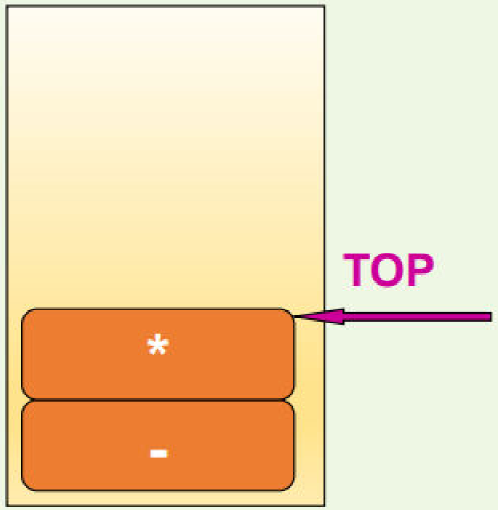
Topstack has less precedence so PUSH to stack

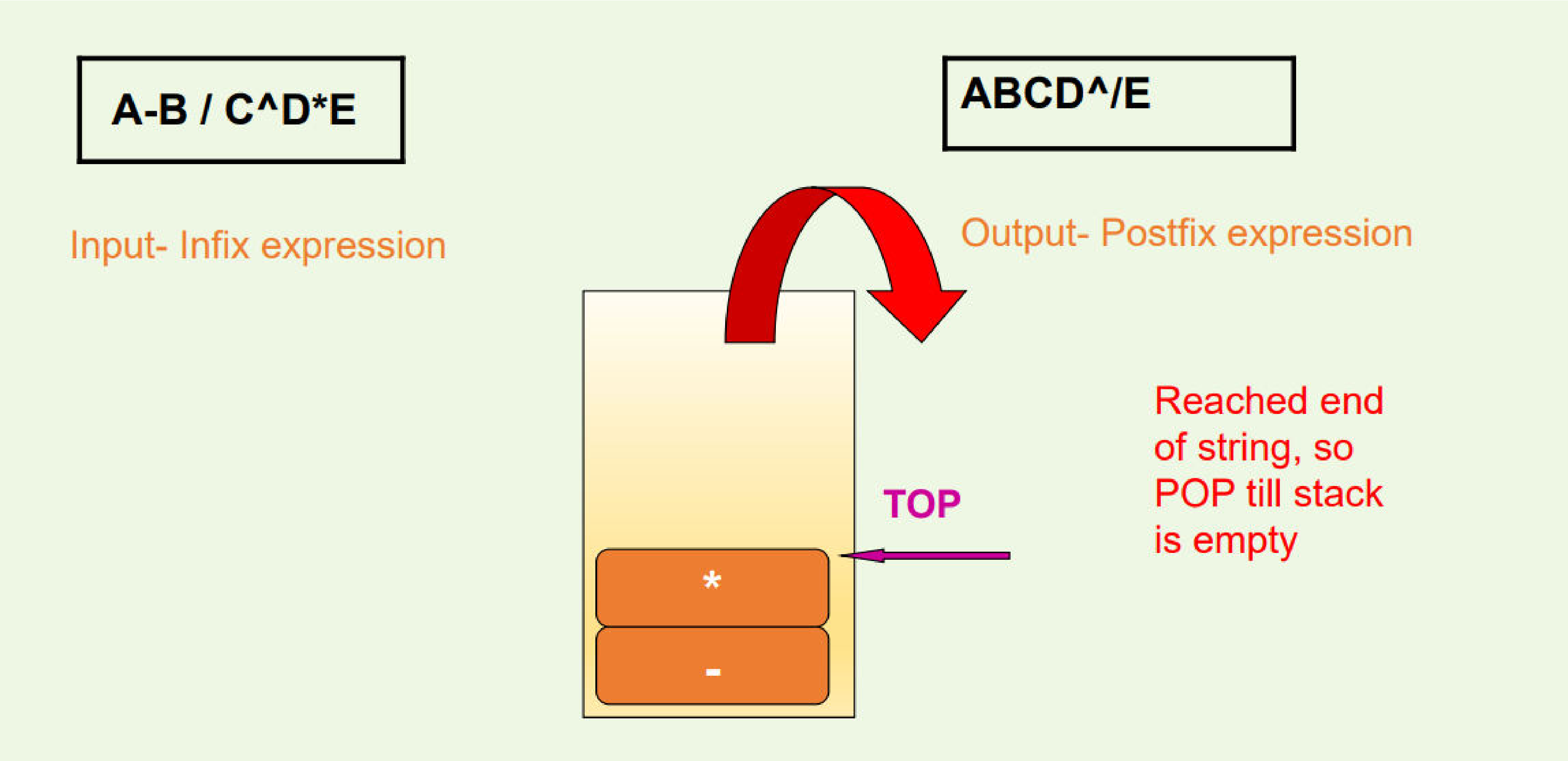
A-B / C^D*E

Input- Infix expression

ABCD^/E

Output- Postfix expression




A-B / C^D*E
ABCD^/E
Input- Infix expression
Output- Postfix expression
Reached end of string, so POP till stack is empty
TOP
*
-

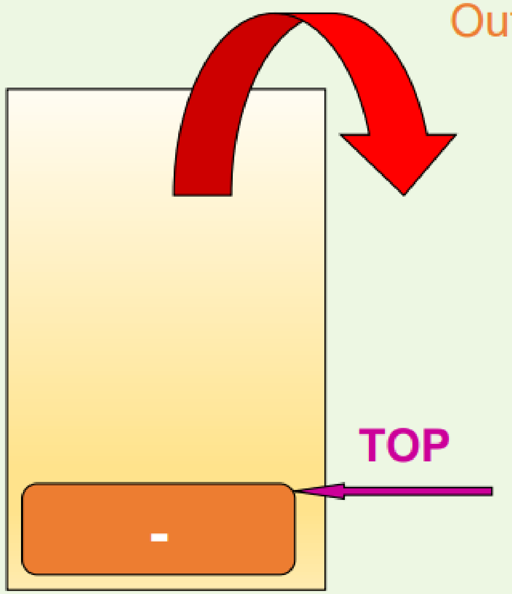
Implementation :

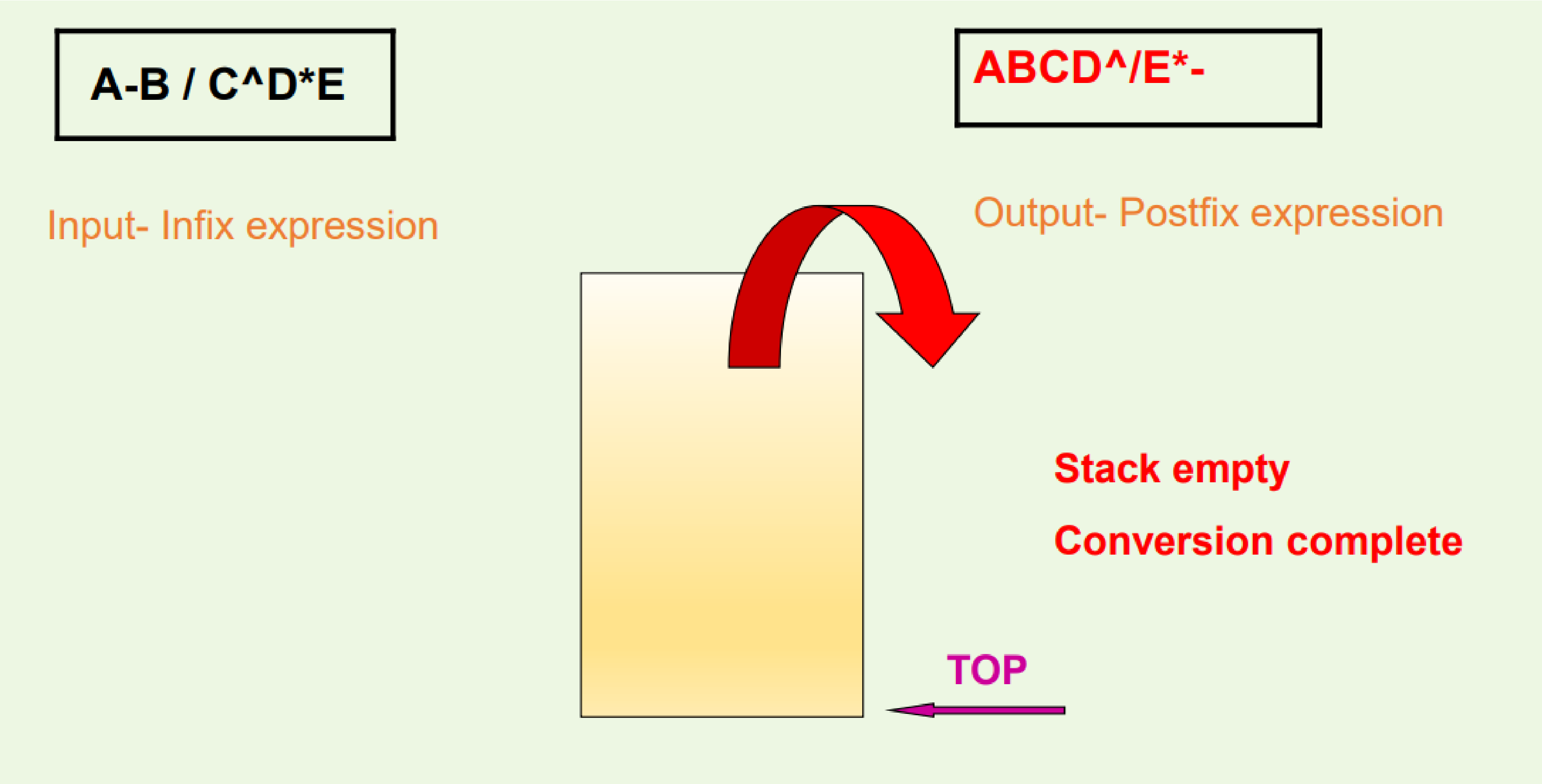
A-B / C^D*E

Input- Infix expression

ABCD^/E*

Output- Postfix expression




A-B / C^D*E
ABCD^/E*-
Input- Infix expression
Output- Postfix expression
Stack empty
Conversion complete
TOP

Convert the following infix expression A + B * C / D - E to postfix

Next Symbol	*Postfix String*	*Stack*	*Rule*
A	A		2
+	A	+	3
B	A B	+	2
*	A B	+ *	3
C	A B C	+ *	2
/	A B C *	+ /	3
D	A B C * D	+ /	2
-	A B C * D / +	-	3
E	A B C * D / + E	-	2
	A B C * D / + E -		6

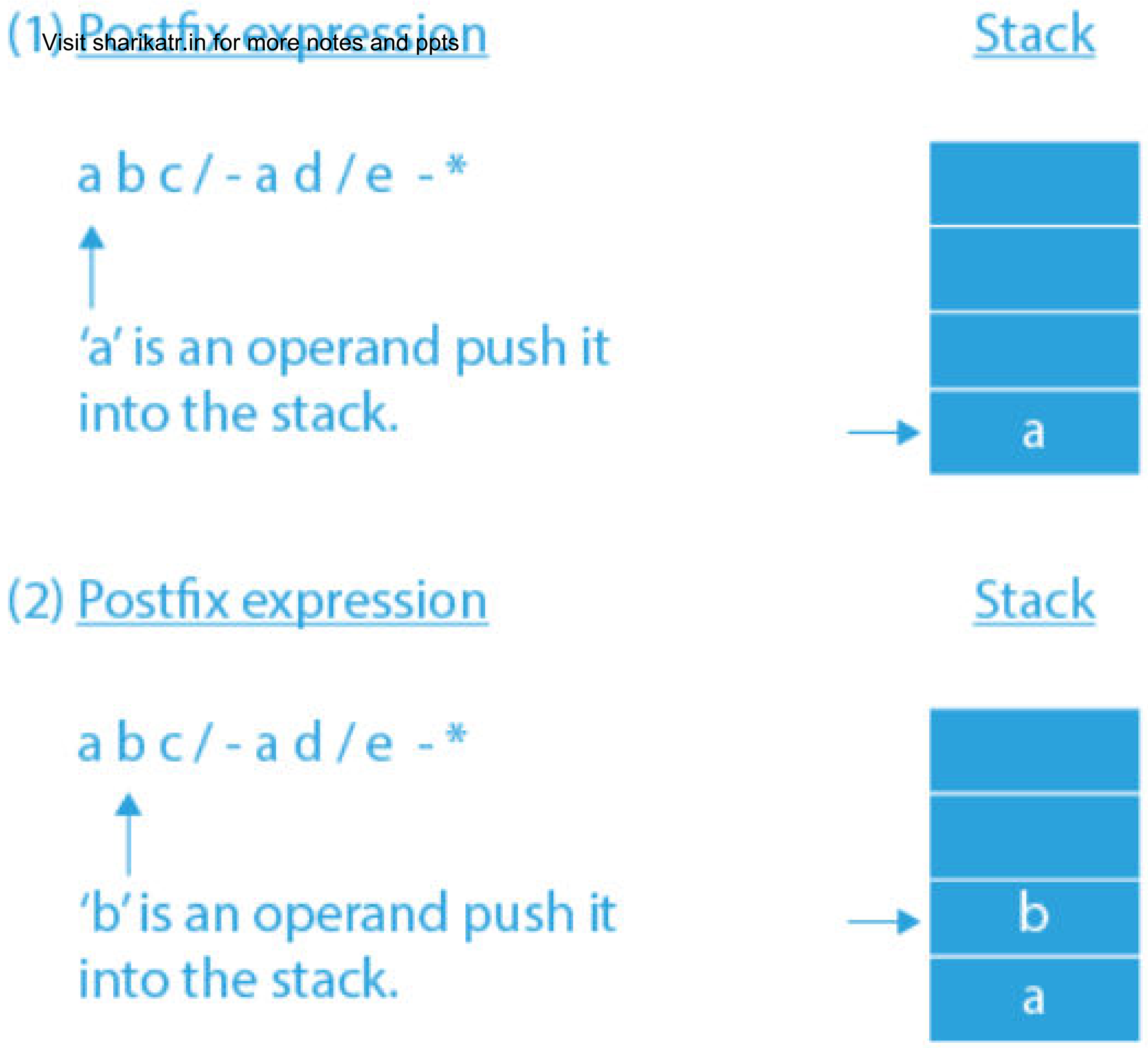
EXERCISE

Find the postfix expression of following infix expression

(A+B)*K+D*(E+F*G)+H

Convert P*(Q+R)/S to infix

Postfix to infix conversion

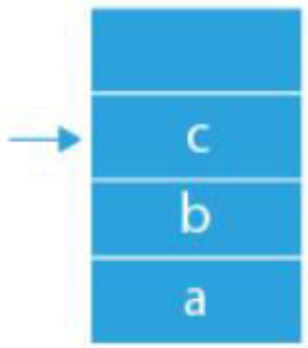


(3) Postfix expression

Stack

a b c / - a d / e - *

'c' is an operand. So, push it into the stack.



(4) Postfix expression

Stack

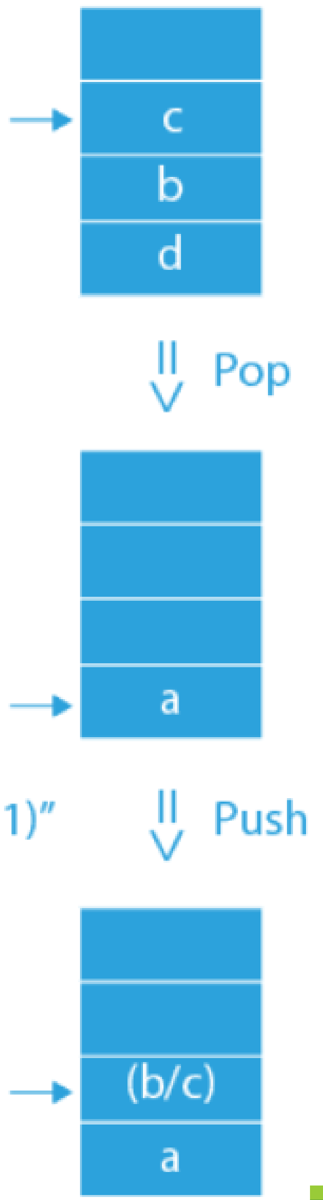
a b c / - a d / e - *

'/' is an operator. Pop two expressions from the stack.

Operand 1 = c
Operand 2 = b
Operator = /

Push
"(Operand 2, Operator , Operand 1)"

=> (b/c)
into the stack.



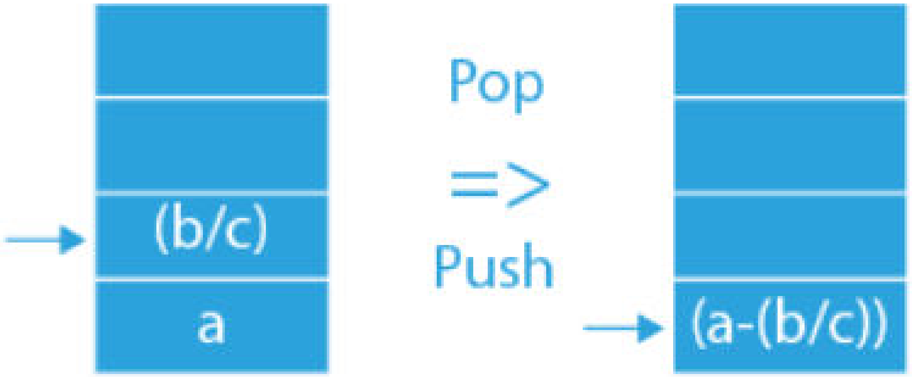
(5) Postfix expression

a b c / - a d / e - *

'-' is an operator. So, operand 1 = (b/c)

Operand 2 = a

push (operand 2 operator operand 1)
i.e, (a - (b / c)) into the stack.

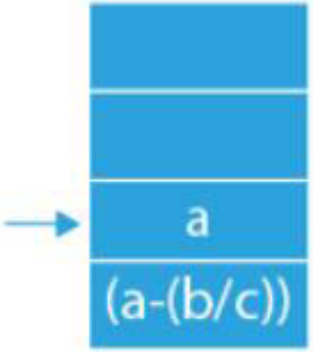


(6) Postfix expression

a b c / - a d / e - *

'a' is an operand. So, push it into the stack.

Stack

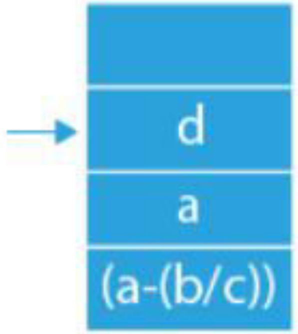


(7) Postfix expression

Stack

a b c / - a d / e - *

'd' is an operand. So, push it into the stack.



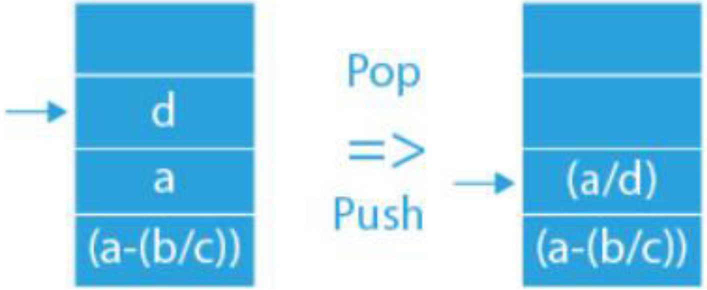
(8) Postfix expression

a b c / - a d / e - *

'/' is an operator.

Operand 1 = d   Operand 2 = a

Push, (operand 2 operator operand 1) i.e (a/d)

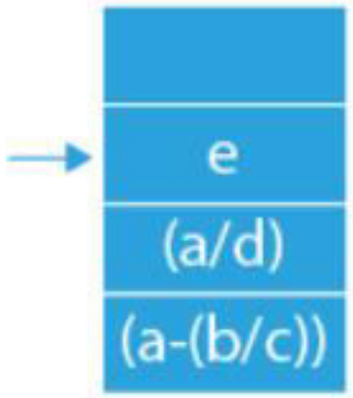


(9) Postfix expression

a b c / - a d / e - *

'e' is an operand. So, push it into the stack.

Stack



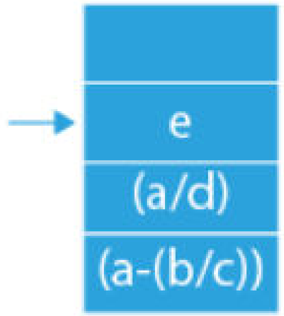
(10) Postfix expression

a b c / - a d / e - *

Operator = -

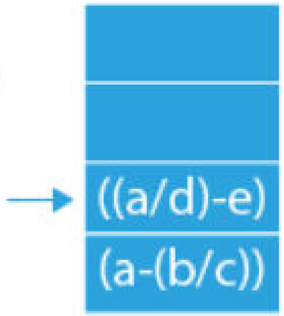
Operand 1 = e   Operand 2 = (a/d)

Push, (operand 2 operator operand 1) i.e ((a/d)-e)



Pop + Push

=>



(11) Postfix expression

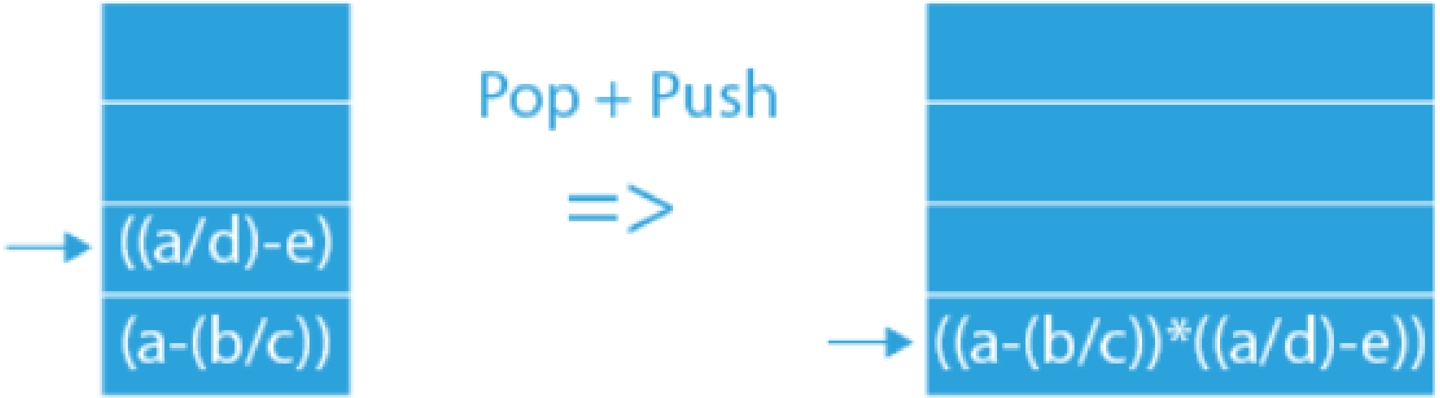
a b c / - a d / e - *

Operator = *

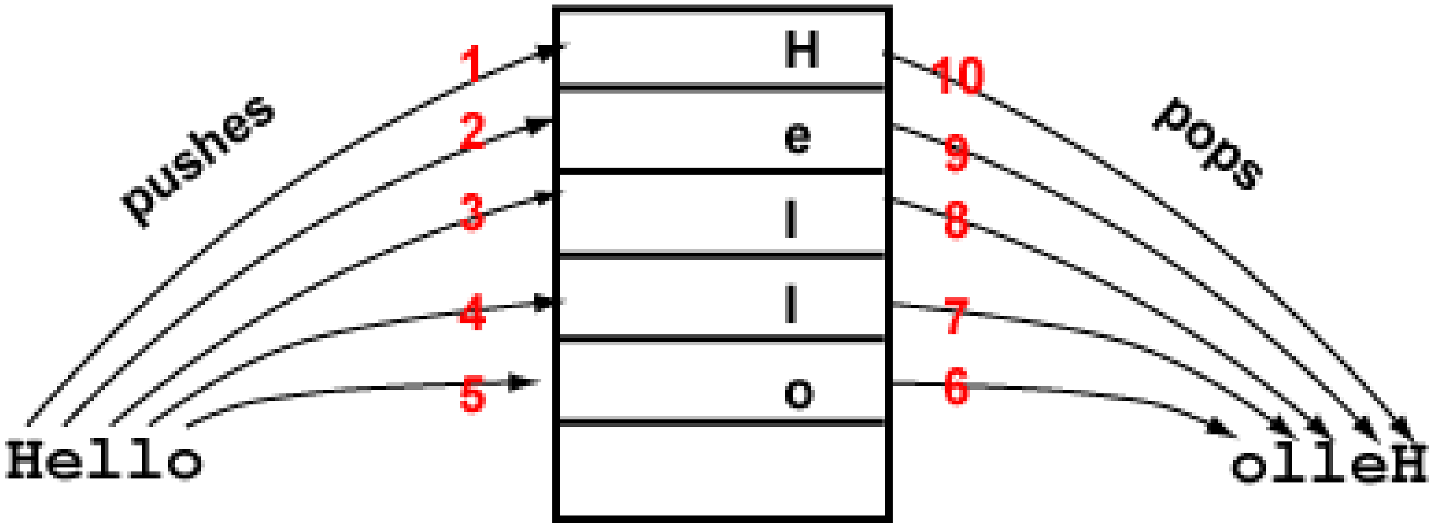
Operand 1 = ((a/d)-e)   Operand 2 = (a-(b/c))

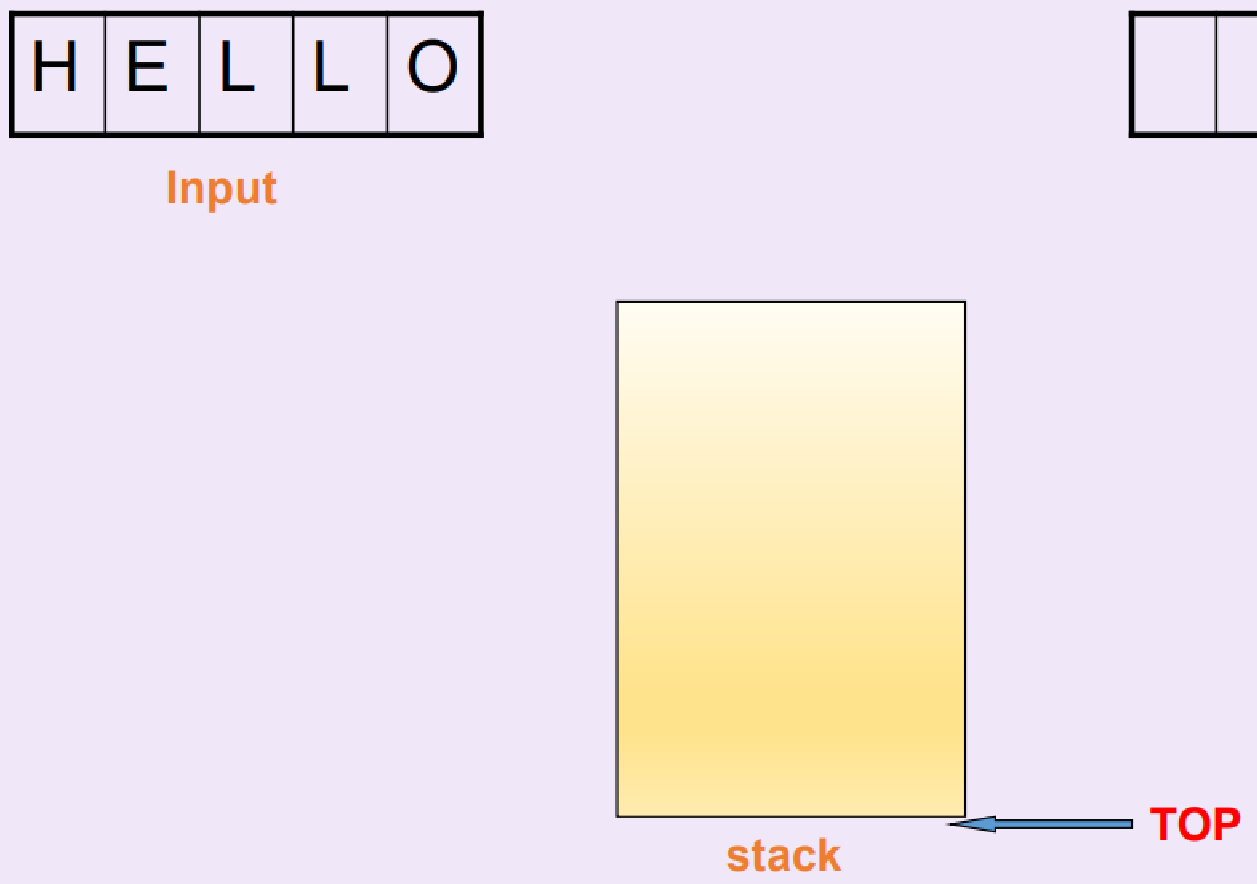
Push, (operand 2 operator operand 1) i.e ((a-(b/c))*((a/d)-e))

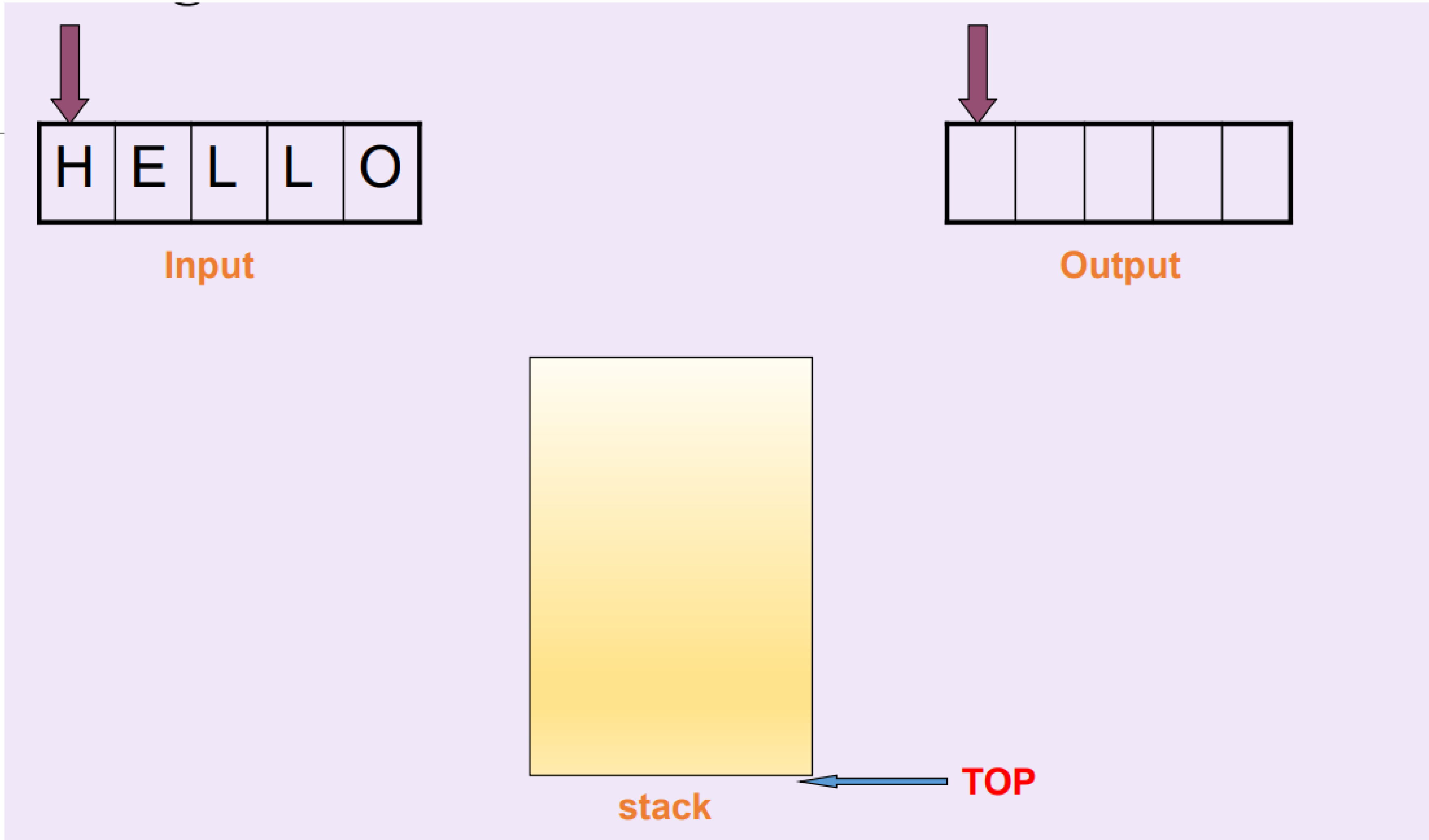
Stack:

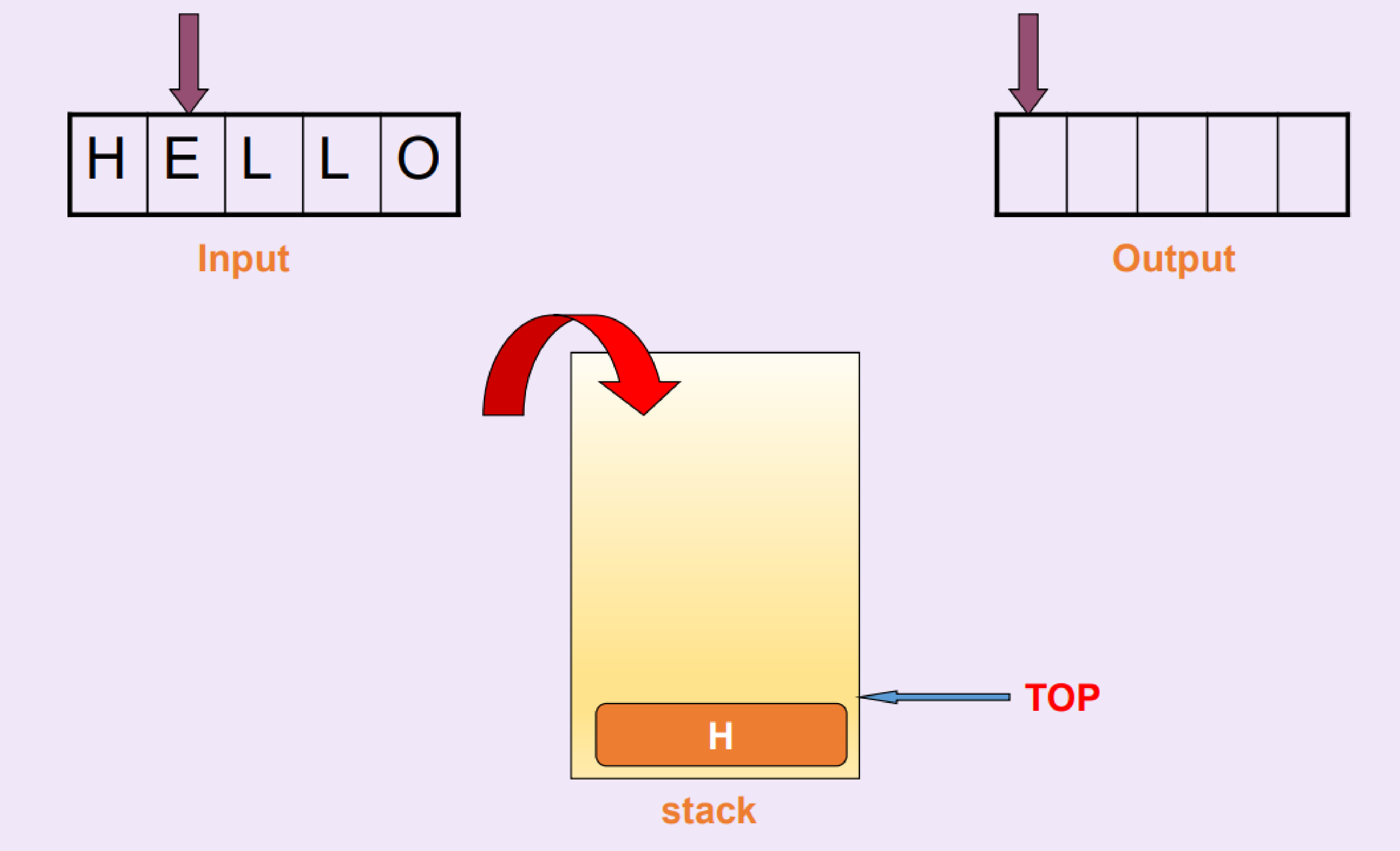


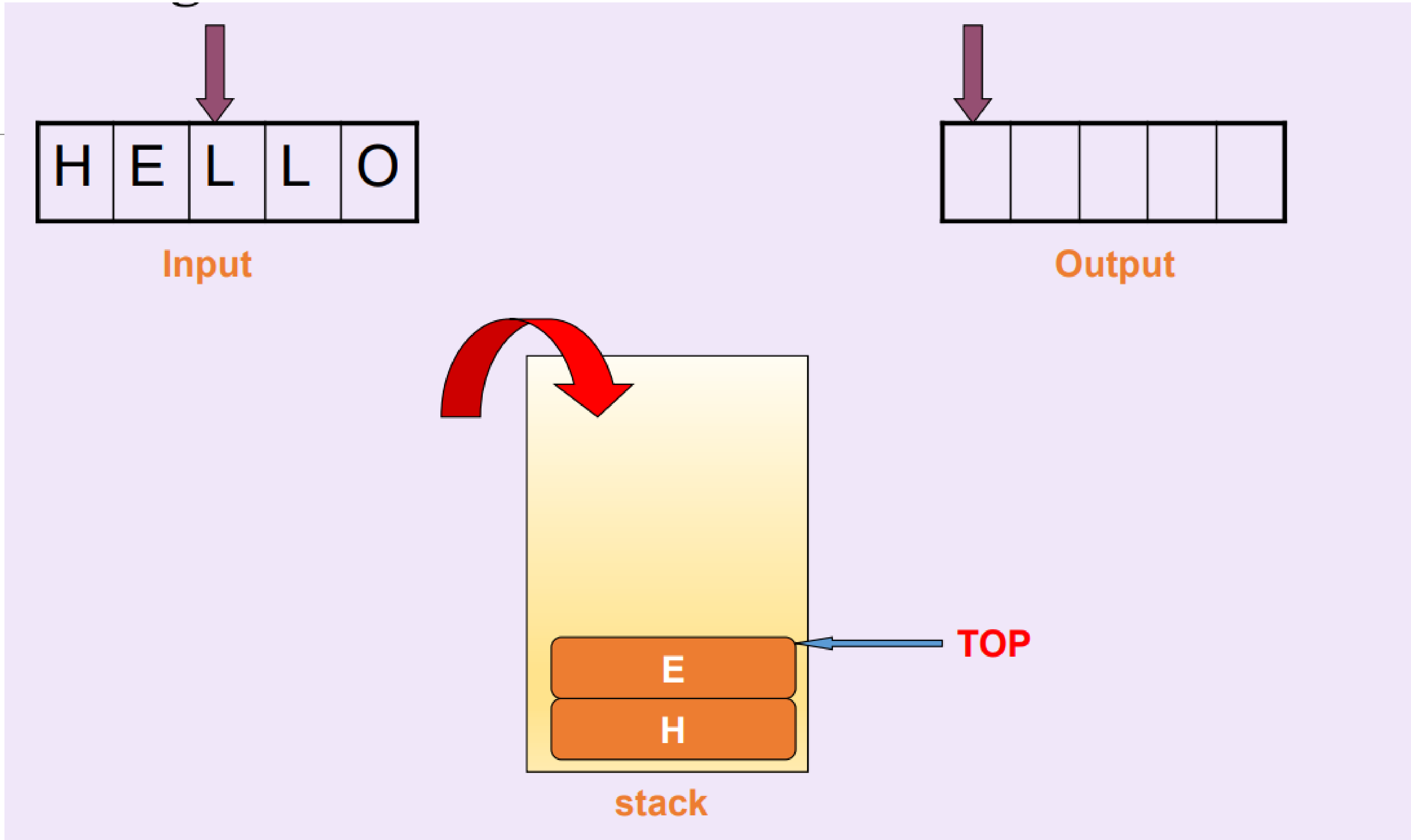
String Reversal

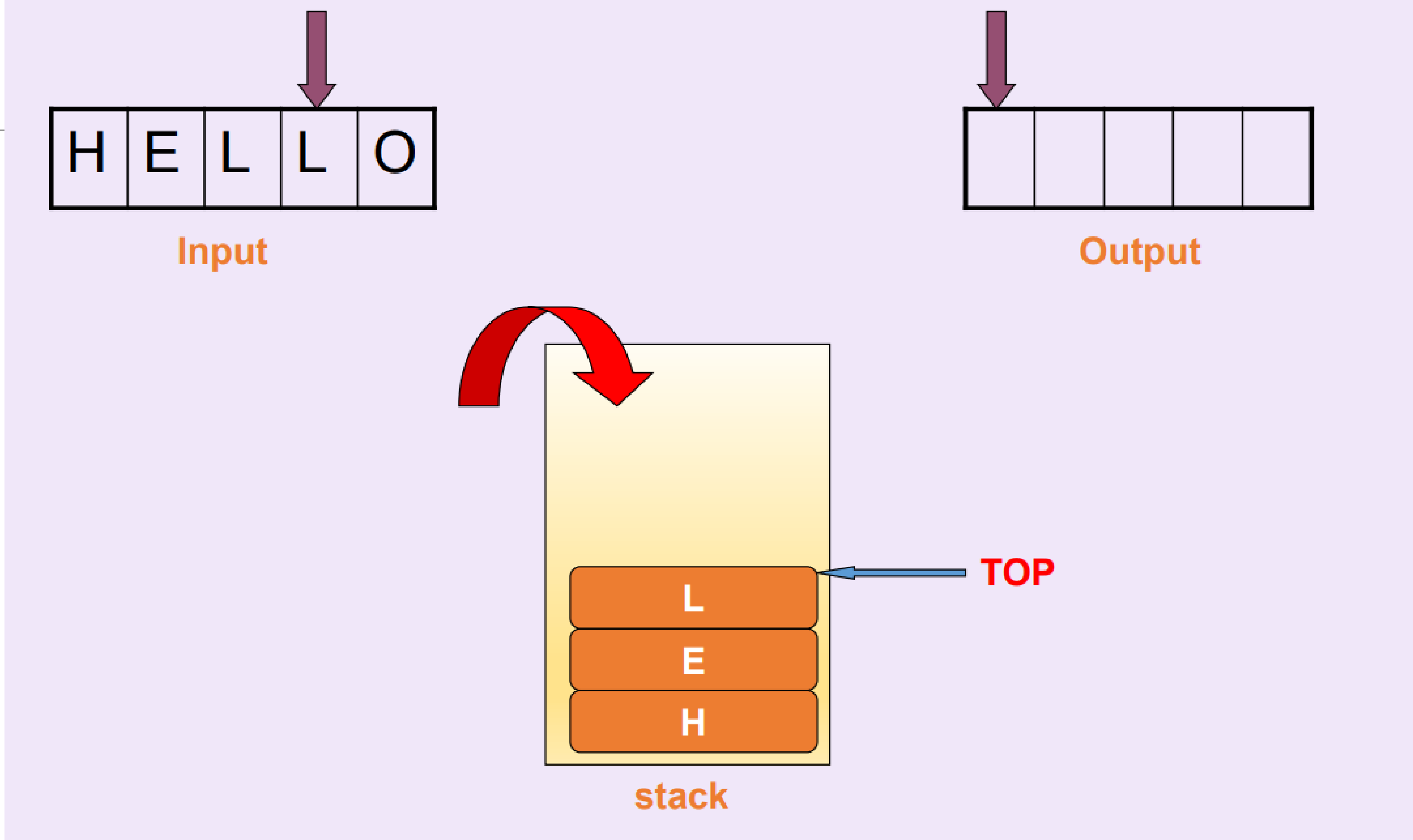


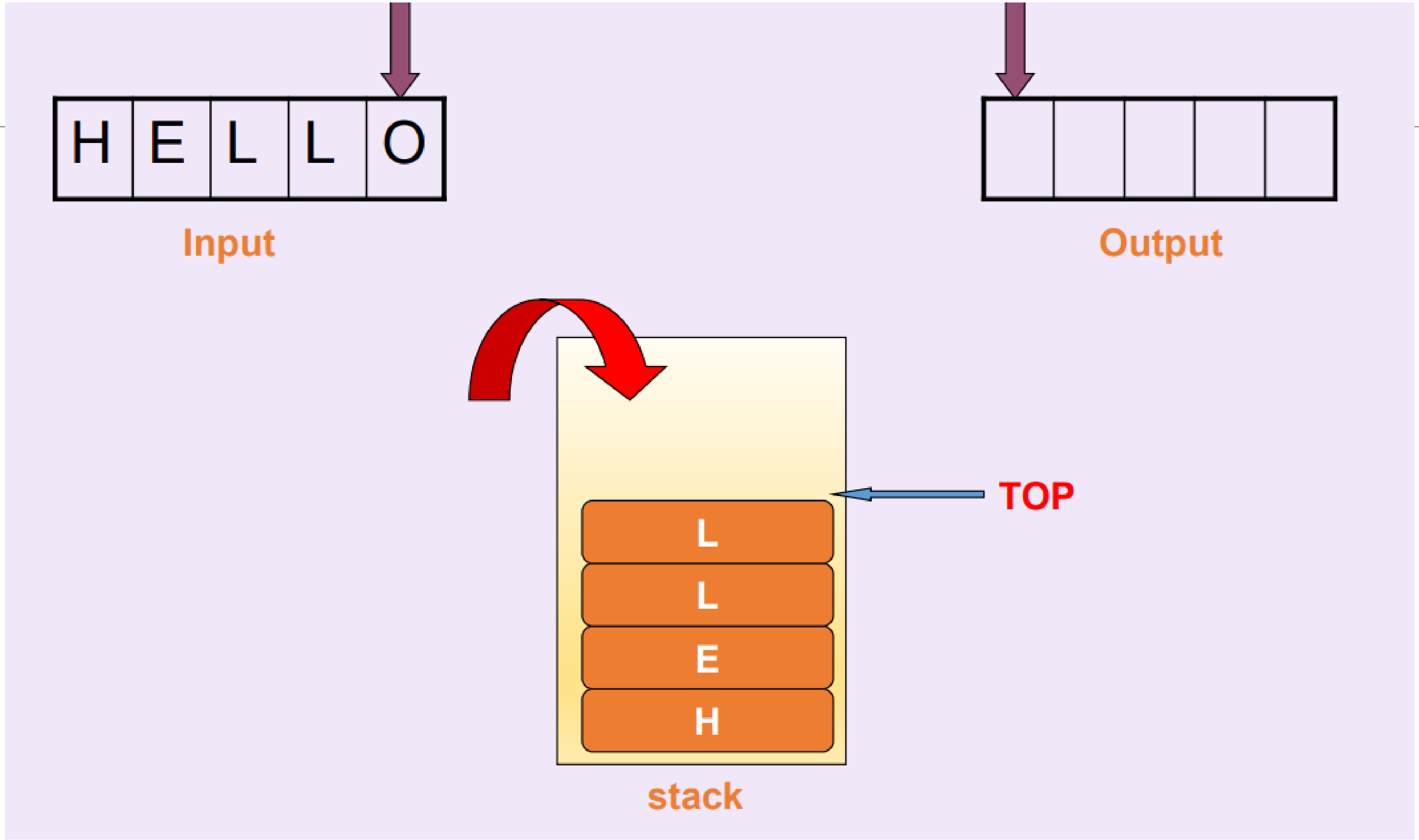

H E L L O
Input
Output
stack
TOP


H
E
L
L
O
Input
Output
stack
TOP

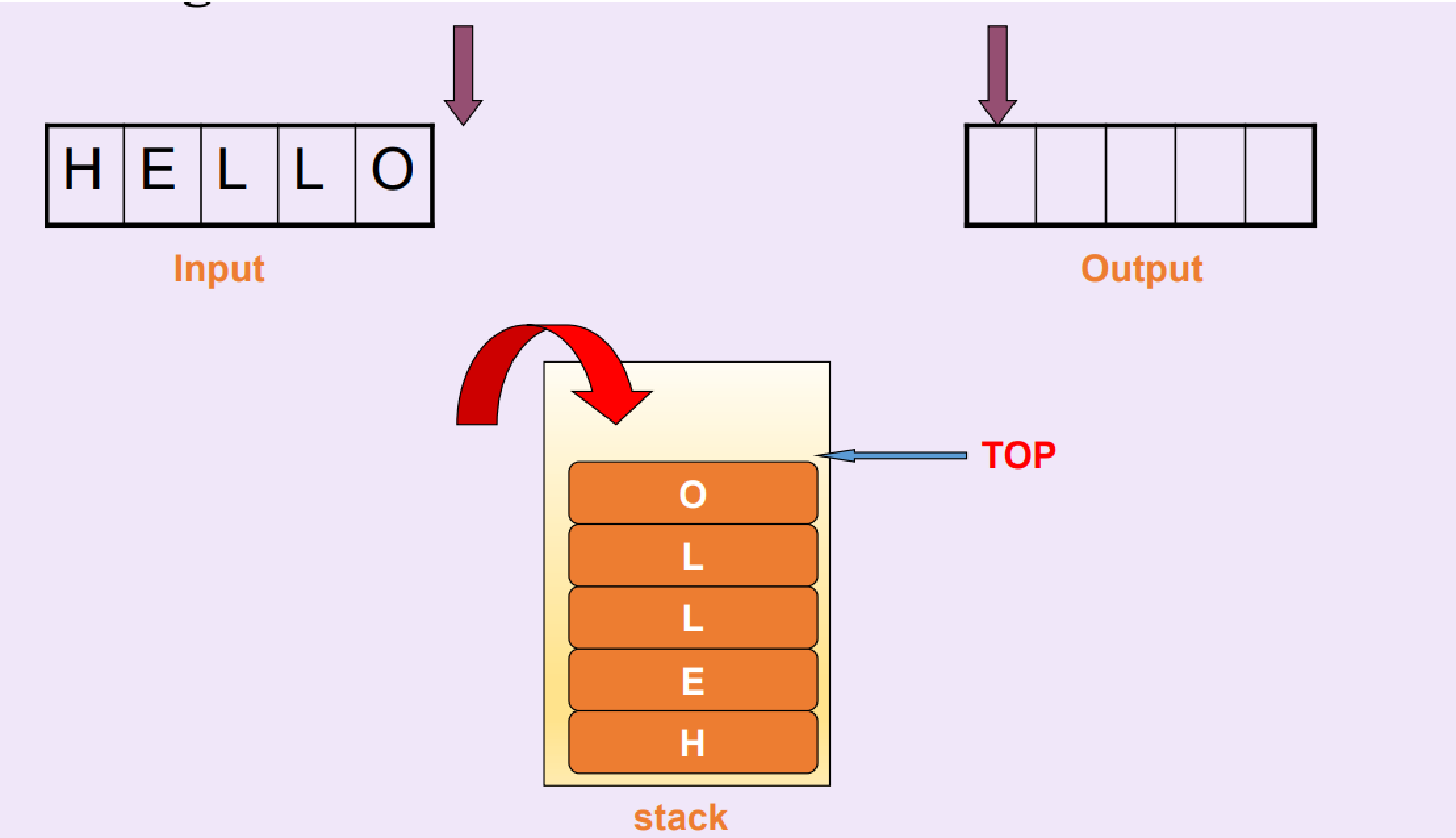

H
E
L
L
O
Input
Output
H
TOP
stack

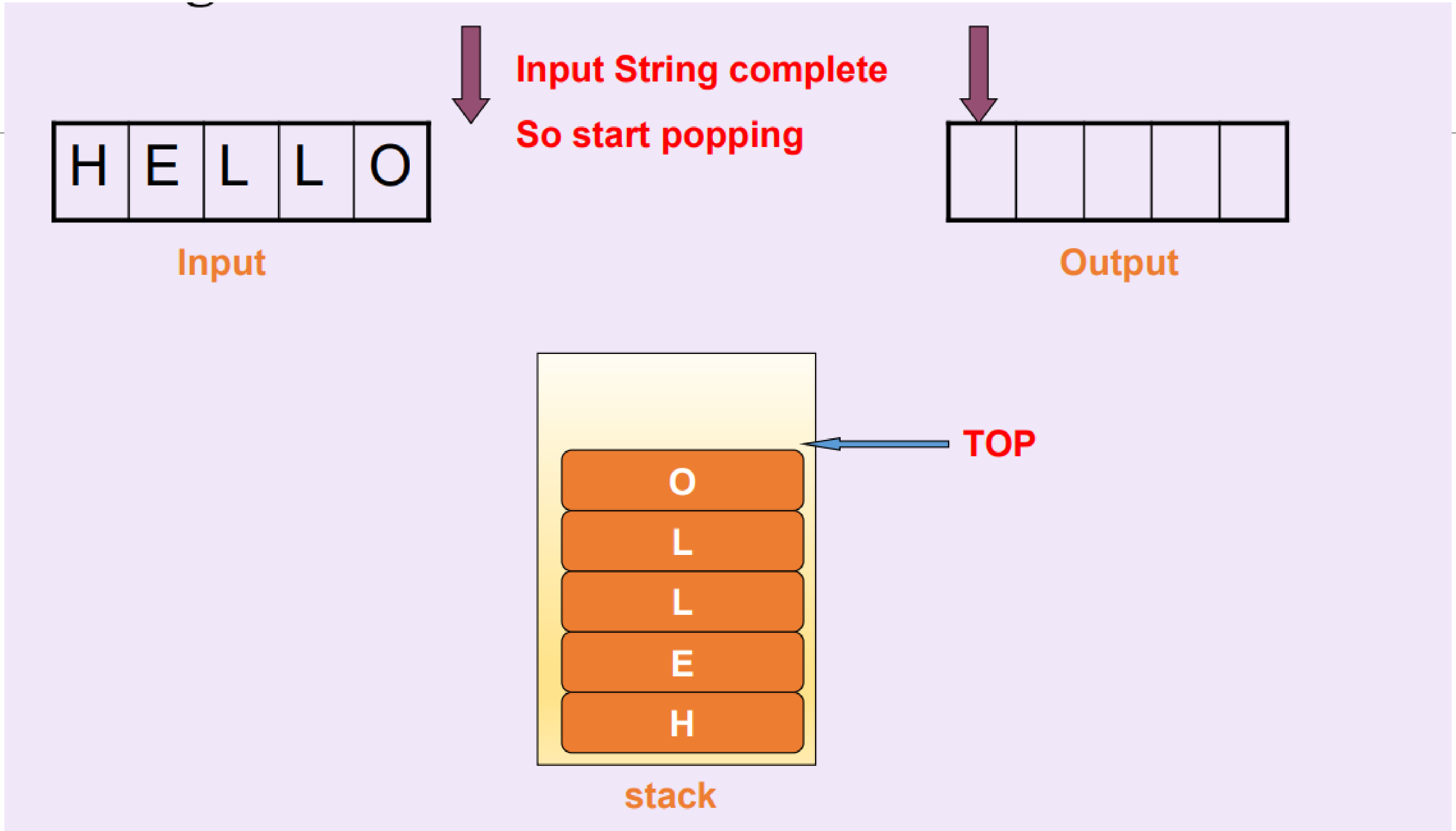

H
E
L
L
O
Input
Output
TOP
E
H
stack

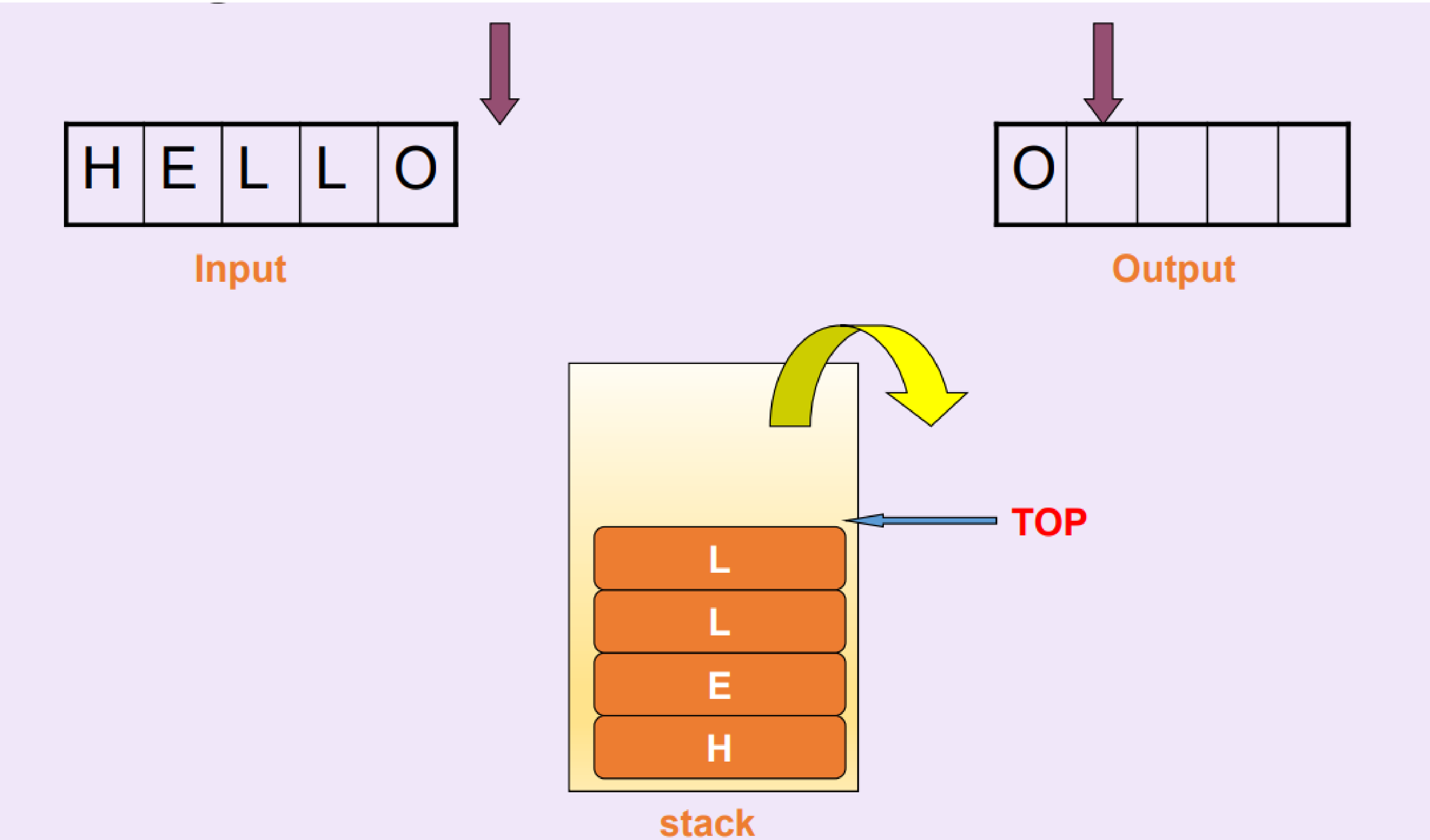

H
E
L
L
O
Input
Output
L
E
H
TOP
stack

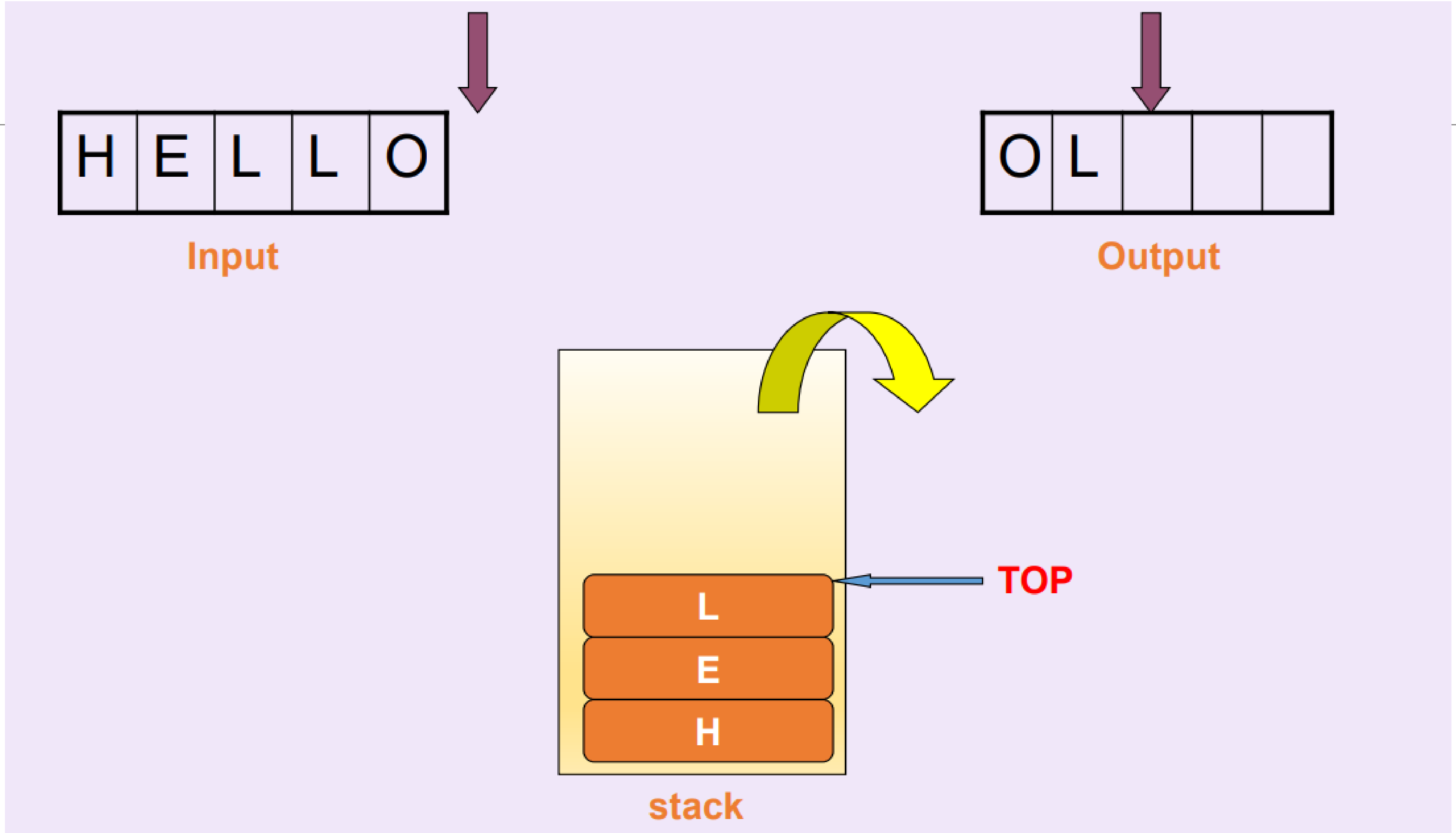


H
E
L
L
O
Input
Output
TOP
L
L
E
H
stack

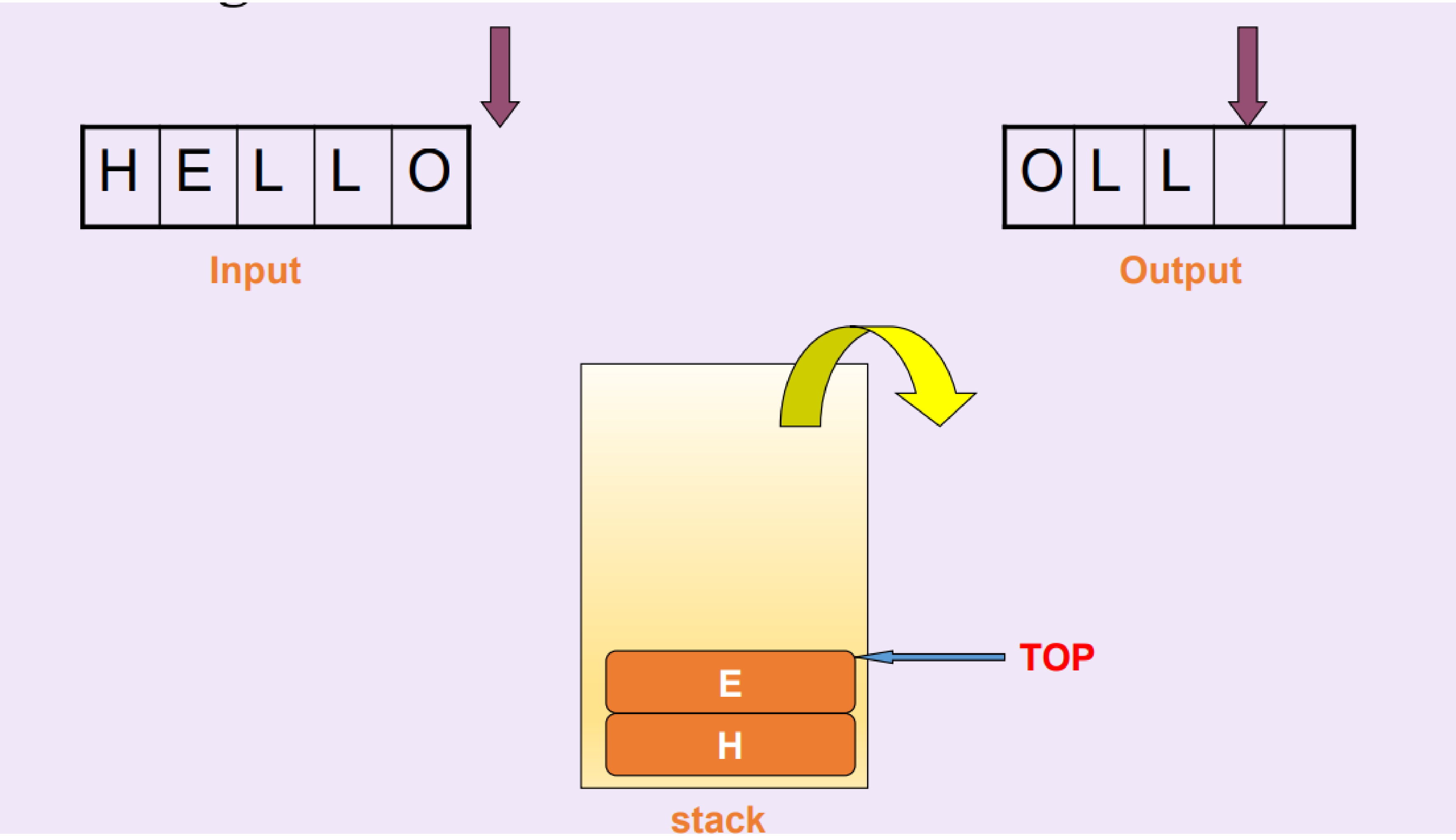

H E L L O
Input
Output
O
L
L
E
H
TOP
stack

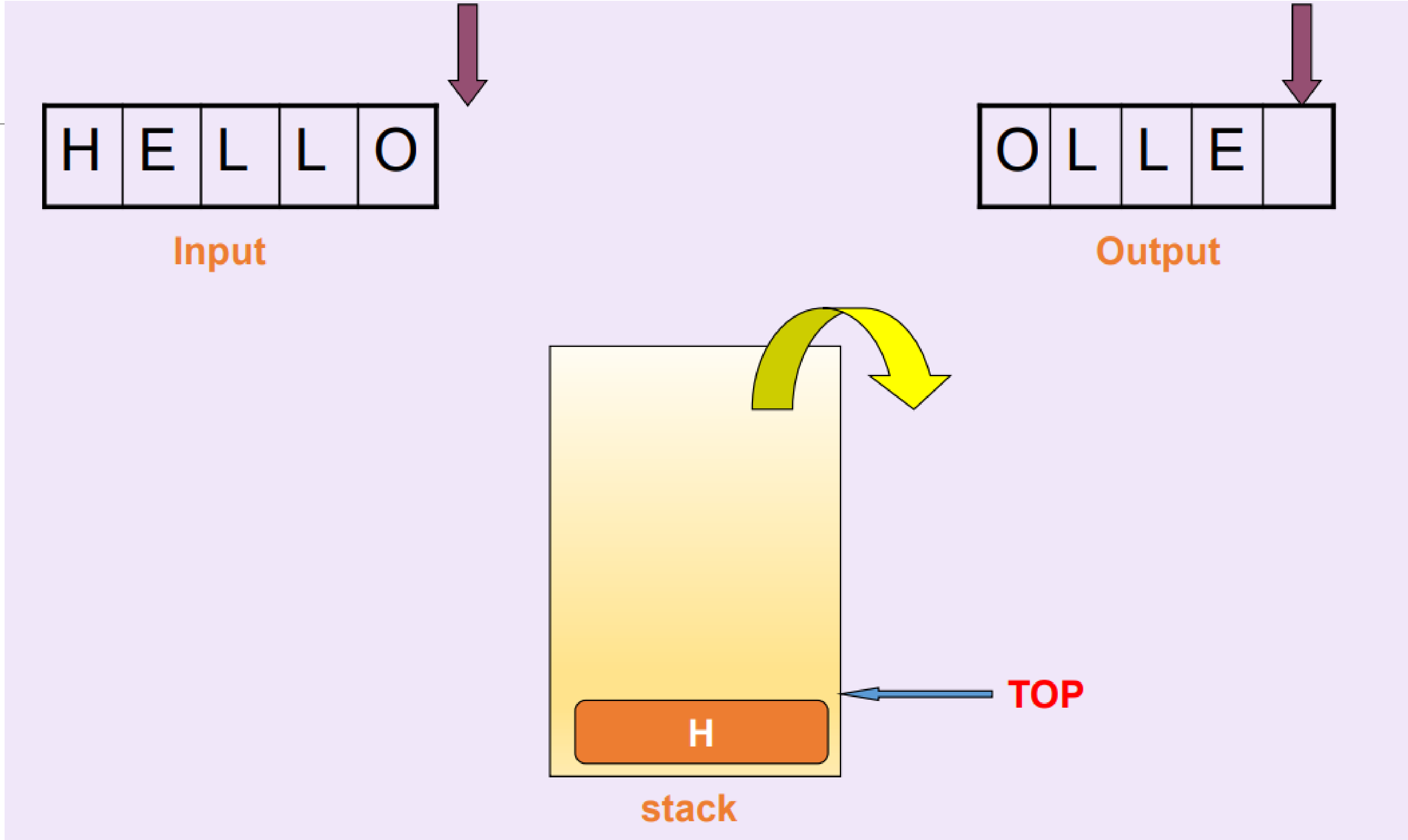

Input String complete
So start popping
H E L L O
Input
Output
TOP
O
L
L
E
H
stack

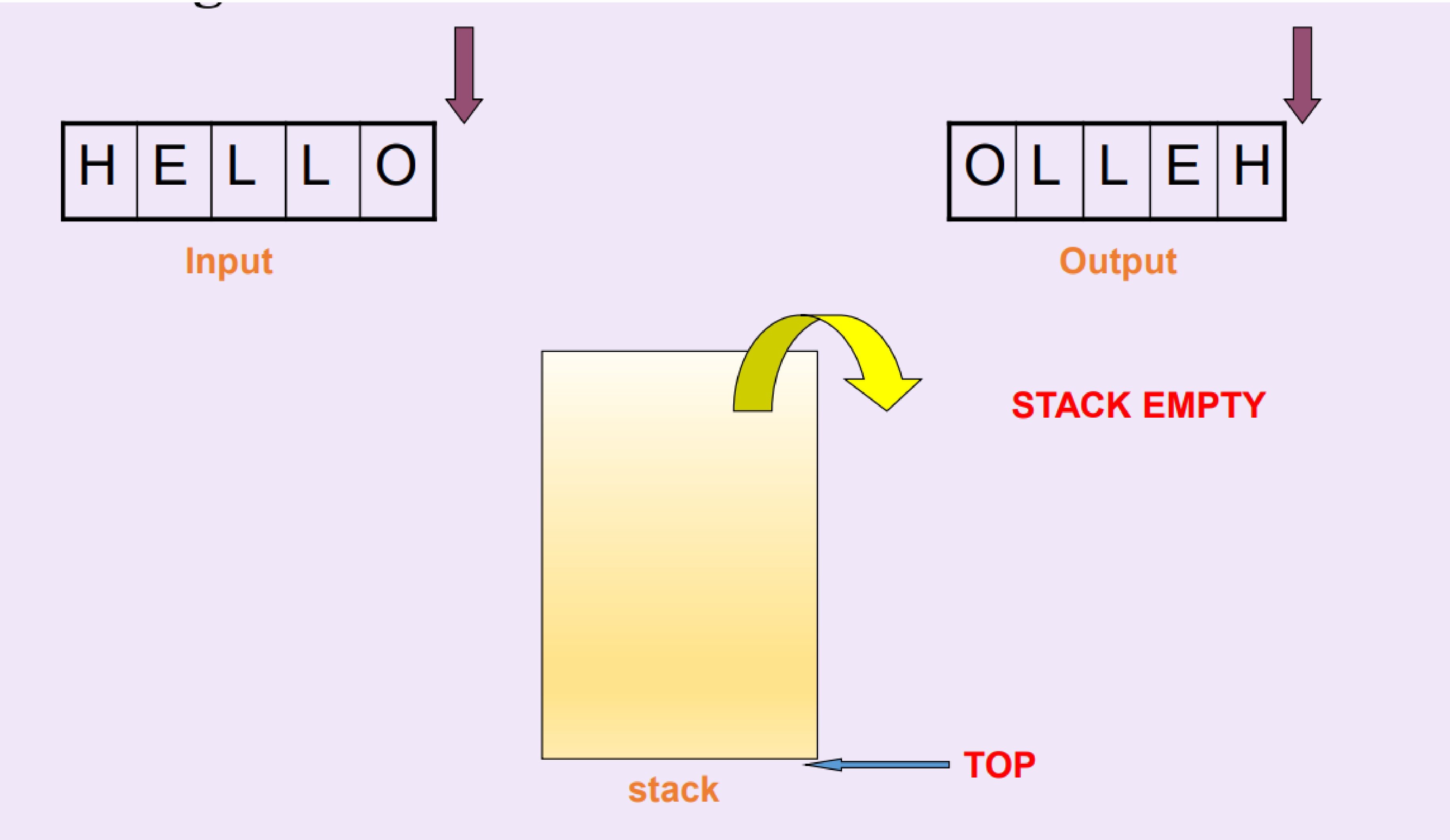

H E L L O
Input
O
Output
TOP
L
L
E
H
stack



H E L L O
Input
O L
Output
L
E
H
TOP
stack


H E L L O
Input
O L L
Output
E
H
TOP
stack


H
E
L
L
O
Input
O
L
L
E
Output
H
TOP
stack


H E L L O
Input
O L L E H
Output
STACK EMPTY
stack
TOP

Queue


LAST IN
LAST OUT
ONE WAY
FIRST IN
FIRST OUT

QUEUE

A queue

- ordered list,
- linear structure
- insertions take place at one end, the rear,
- And deletions take place at the other end, the front.

Two operations on the queue are

- Insertion (ENQUEUE): Take place at the end called REAR
- Deletion (DEQUEUE): Take place at other end called FRONT

Restrictions on queue

- the first element which is inserted into the queue will be the first one to be removed.
- queues are known as First In First Out (FIFO) lists.

Representation of Queues

Two ways to represent a queue in memory

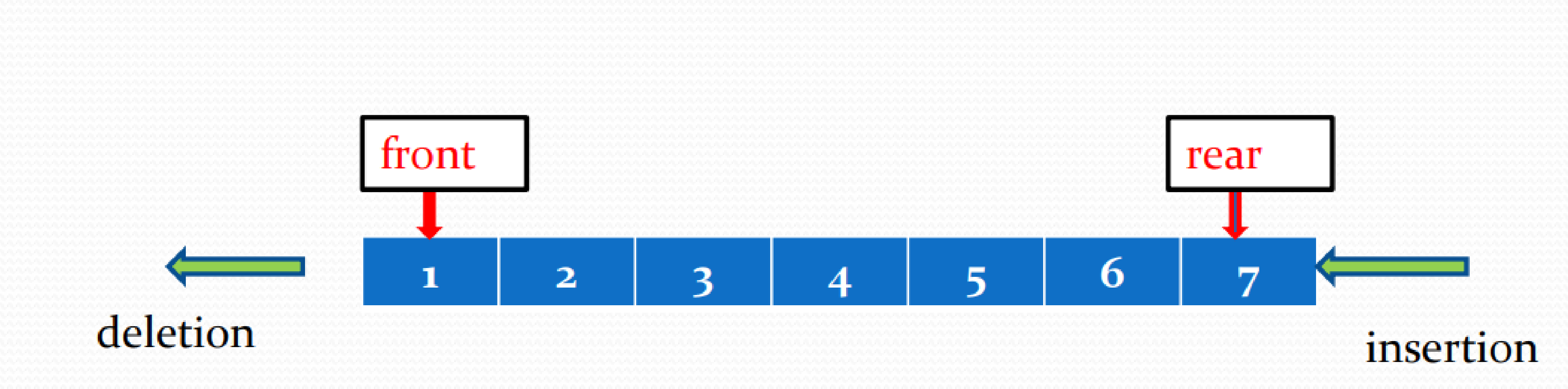
- Using an Array
- Using Linked List

Queue using Array

One dimensional array, say Q[0 … n-1] can be used to represent a Queue.

Two pointers, FRONT and REAR indicate two end of the queue.

Insertion to the REAR end and deletion from FRONT end.



Operations on Queue : Insertion(ENQUEUE)

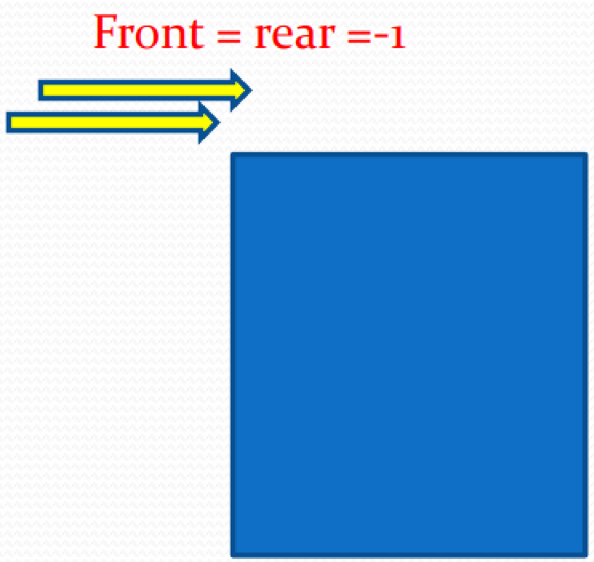
Initially the queue will be initialized as front= -1 and rear = -1

Before inserting check whether the queue is full or not.

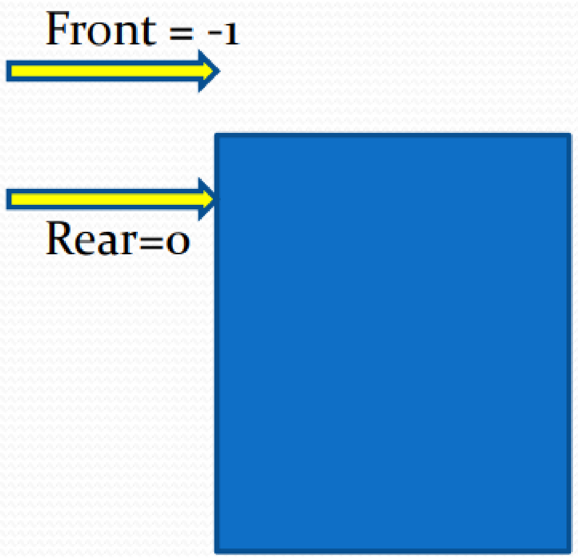
If not full, then insert the element to (REAR+1)

Make sure that the front always points to the first element by incrementing the front pointer when the first element is inserted.

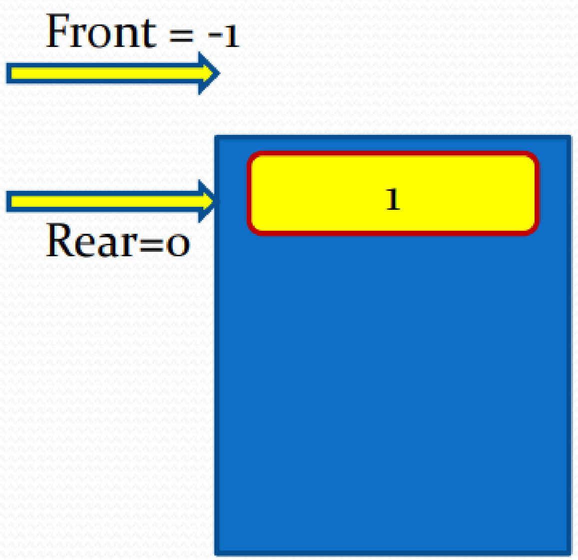
Initially, front = rear =-1



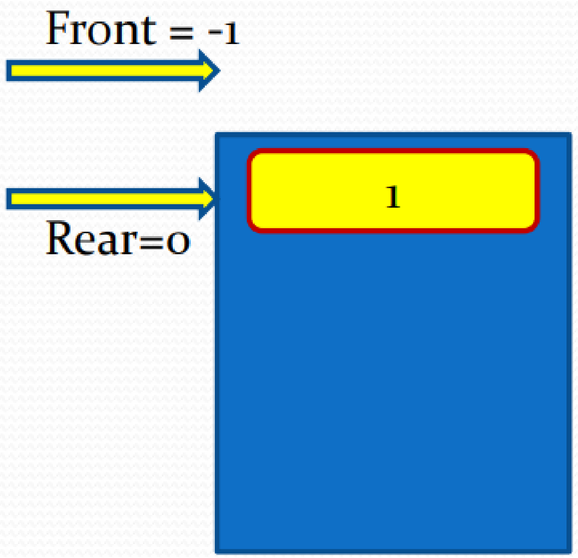
- Insert 1



- Insert 1

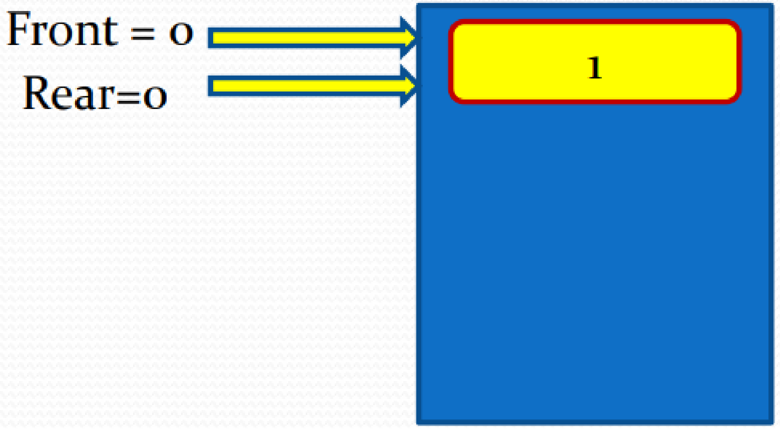


- Insert 1



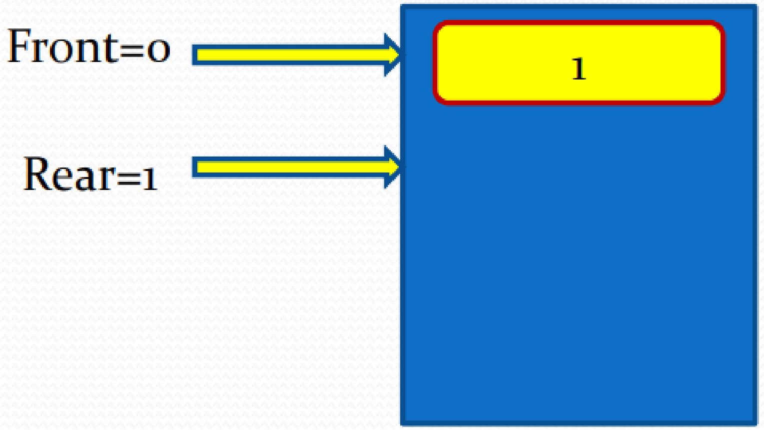
- Change front, so that it points to the first element

- Insert 1

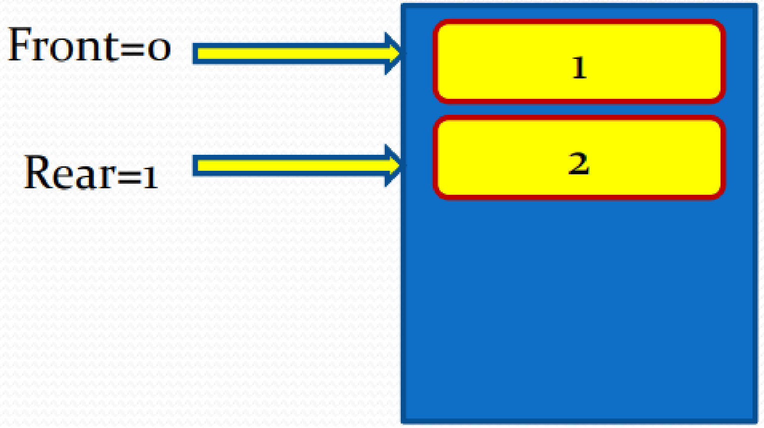


- Change front, so that it points to the first element

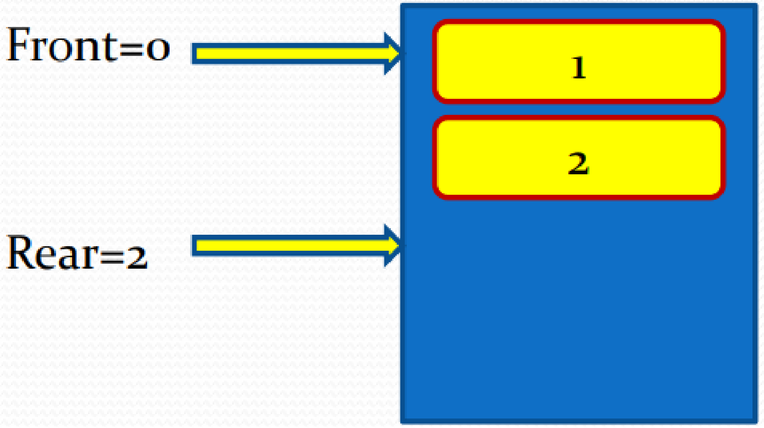
- Insert 2



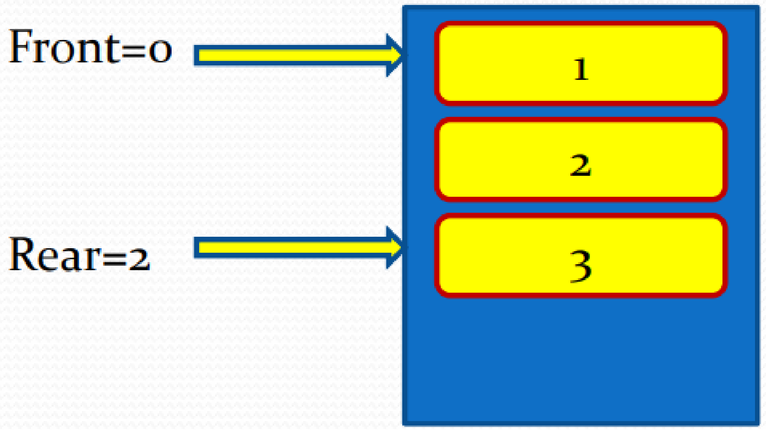
- Insert 2



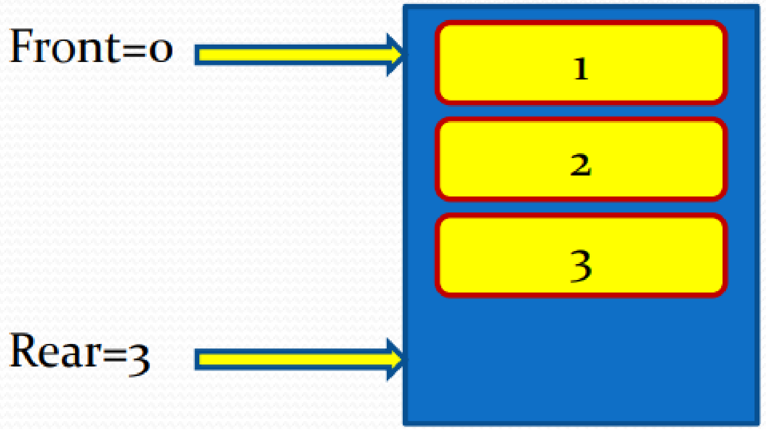
- Insert 3



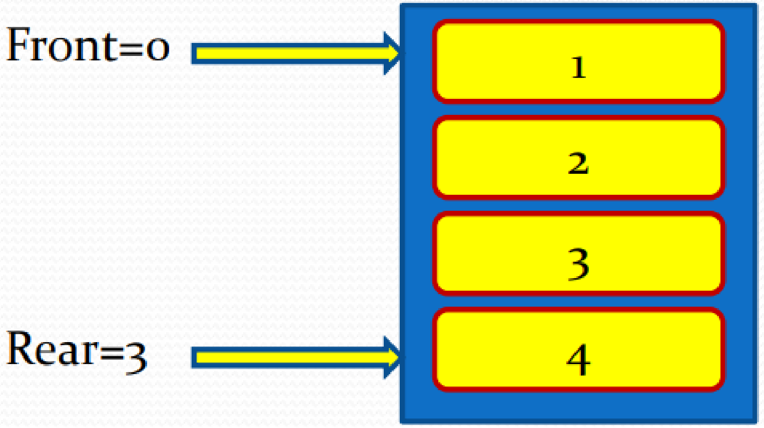
- Insert 3



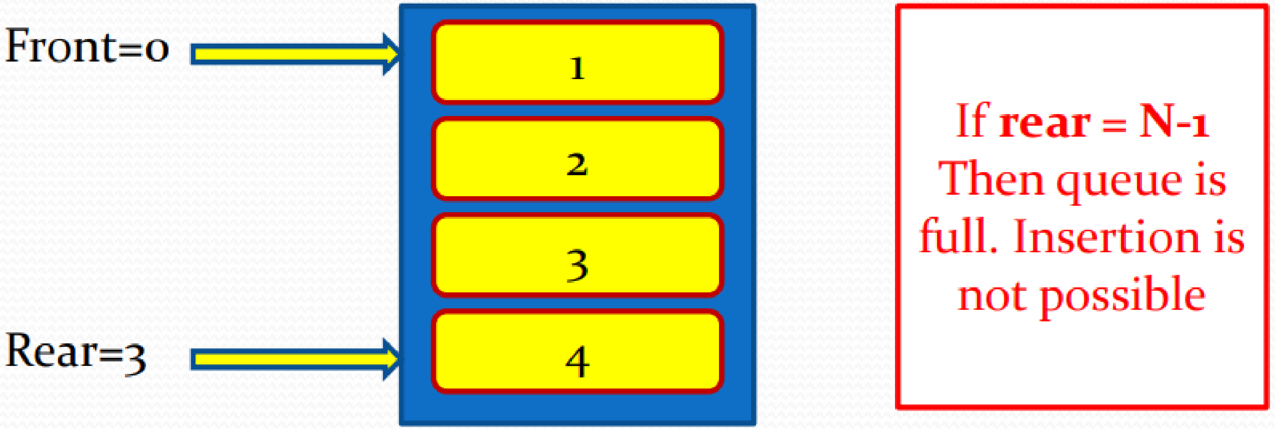
- Insert 4



- Insert 4



- Insert 5


Front=0
Rear=3
1
2
3
4
If **rear = N-1**
Then queue is
full. Insertion is
not possible

Steps:

Algorithm ENQUEUE (ITEM)

Input: ITEM has to be inserted in to the REAR end of the queue.

Output: Queue is enriched with new element ITEM.

Data Structure: Queue is implemented using array.

1. If(REAR=N-1) then
   1. print " Queue is full"
2. Else
   1. If(REAR=-1 AND FRONT=-1) //Queue is empty
      1. Set FRONT=0
   2. EndIf
   3. REAR=REAR+1
   4. Q[REAR]=ITEM
3. EndIf
4. Stop

Deletion(DEQUEUE)

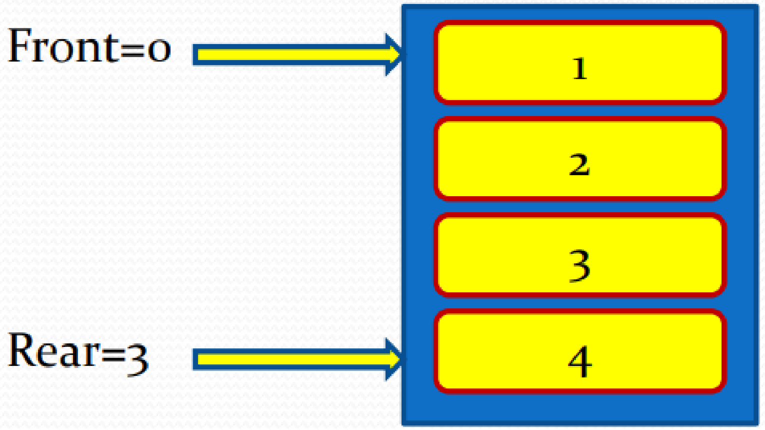
Before deleting, check whether the queue is empty or not.

If not empty, then delete the element

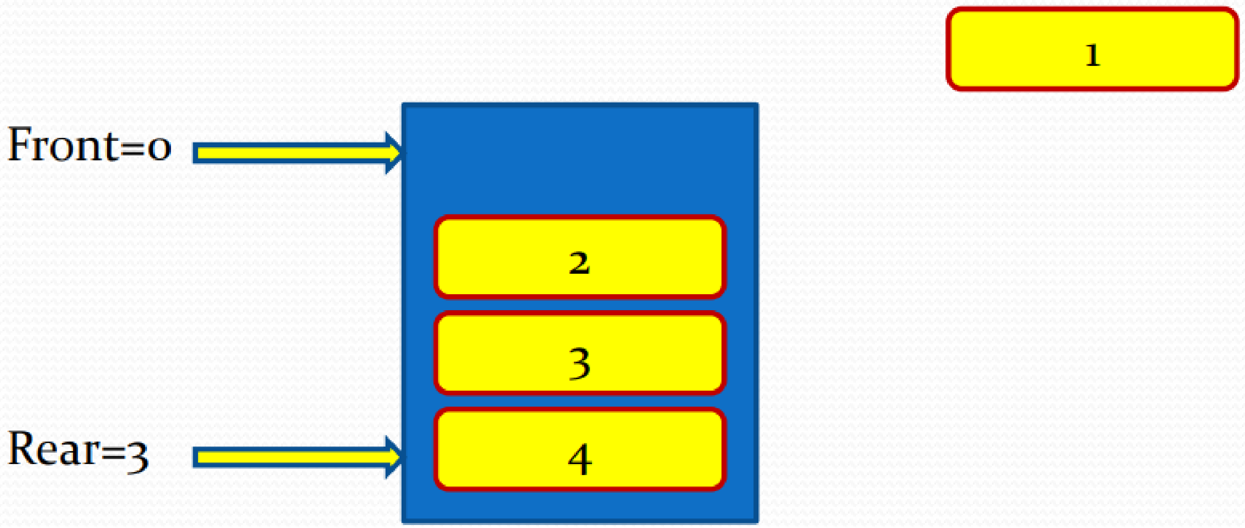
Make sure that the FRONT and REAR always points to -1 by decrementing the pointers when the last element is deleted

- Check whether queue is empty ?
  - Rear>=front and front != -1

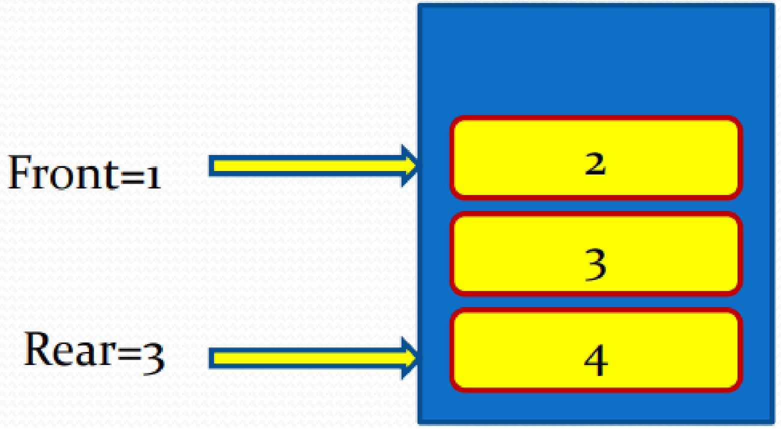
NOT EMPTY !!



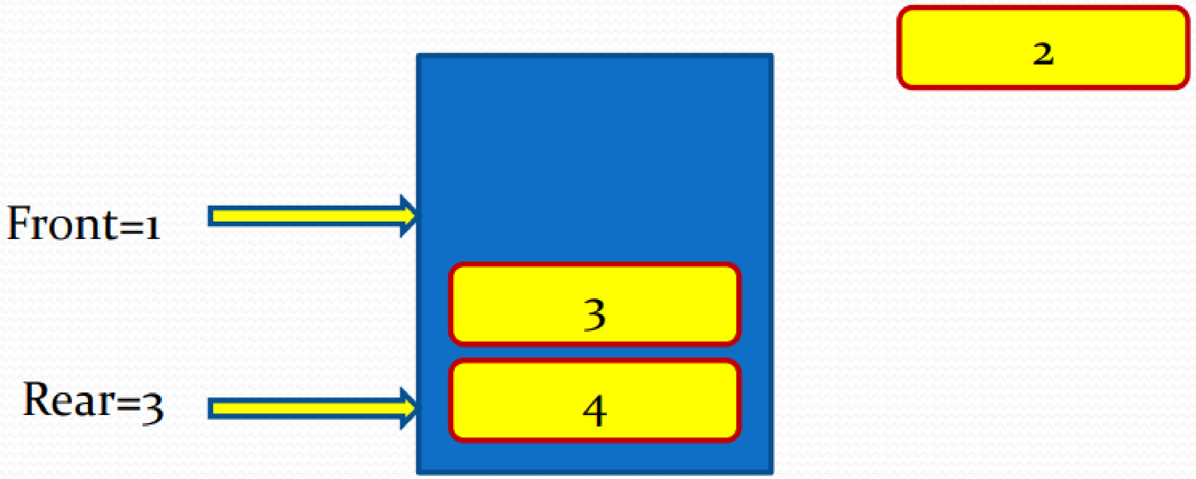
- Dequeue



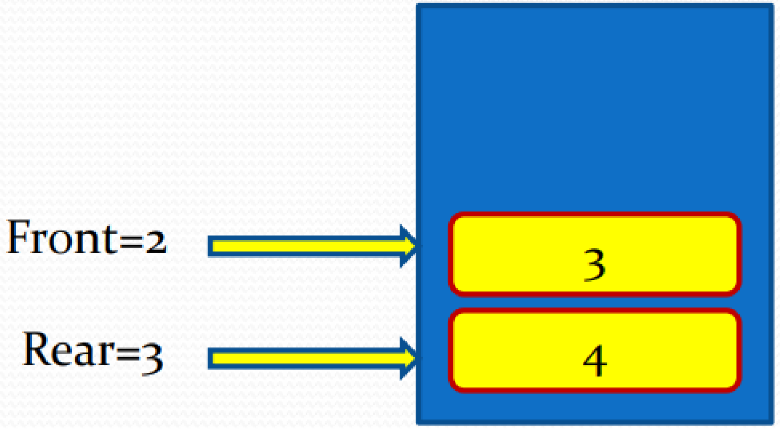
- Dequeue



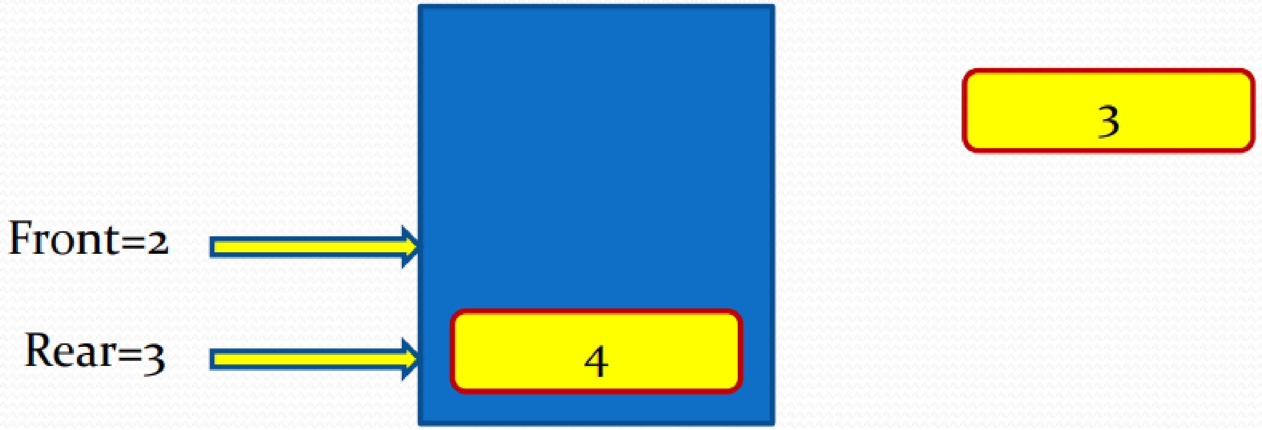
- Dequeue



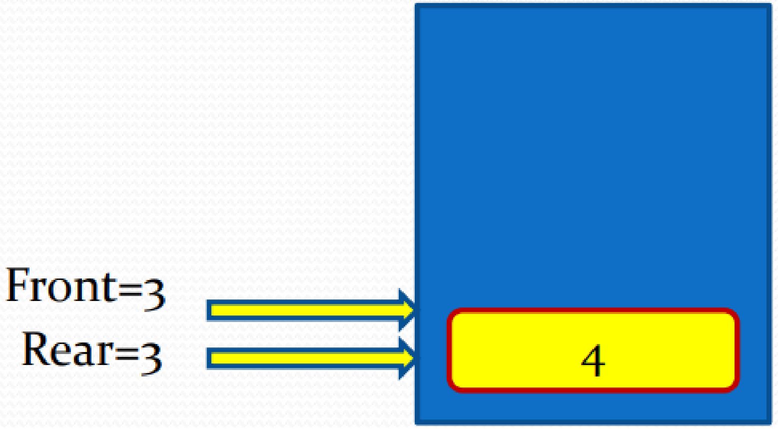
- Dequeue



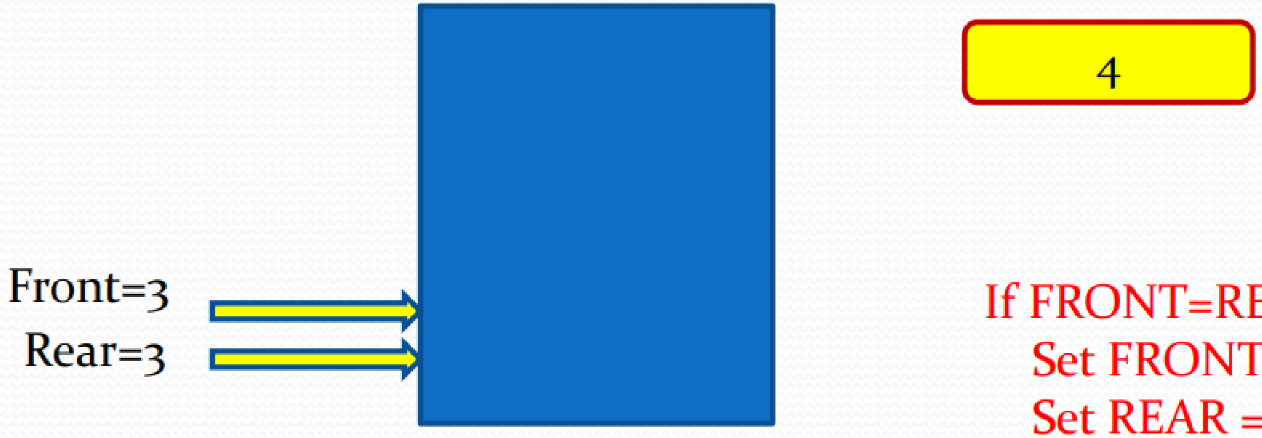
- Dequeue



- Dequeue



- Dequeue

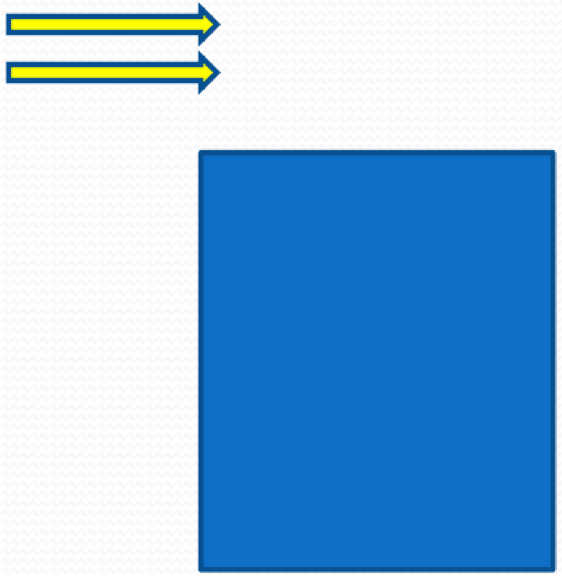


If FRONT=REAR
Set FRONT = -1
Set REAR = -1

- Dequeue

Front= -1

Rear= -1



If FRONT=REAR

Set FRONT = -1

Set REAR = -1

Steps:

Algorithm DEQUEUE()

Input: A Queue with elements and two pointers FRONT and REAR

Output: The deleted element is stored in the ITEM.

Data Structure: Queue is implemented by using array.

1. If(FRONT=-1)
   1. Print " Queue is empty"
2. Else
   1. ITEM=Q[FRONT]
   2. If(FRONT=REAR)
      1. REAR=-1
      2. FRONT=-1
   3. Else
      1. FRONT=FRONT+1
   4. EndIf
3. EndIf
4. Stop

Limitation

Disadvantages of the above implementation is

- As we delete element from the queue, the queue moves down array.
- So the storage space in the beginning is discarded and never used again

Solution

1) Keep FRONT always at the zero index position.

- To maintain front at zero index position, every delete operation would require shifting of all succeeding element in the array by one position
- Advantages:
  - It enables us to utilize all the empty position in an array i.e. no wastage of space.
- Disadvantages:
  - Every delete operation requires shift all the succeeding elements in the queue by one position . If the queue is lengthy, this can be very time consuming

Different Queue Structure

1. Circular Queue
2. DEQue
3. Priority Queue

Circular Queue

Problem with linear queue

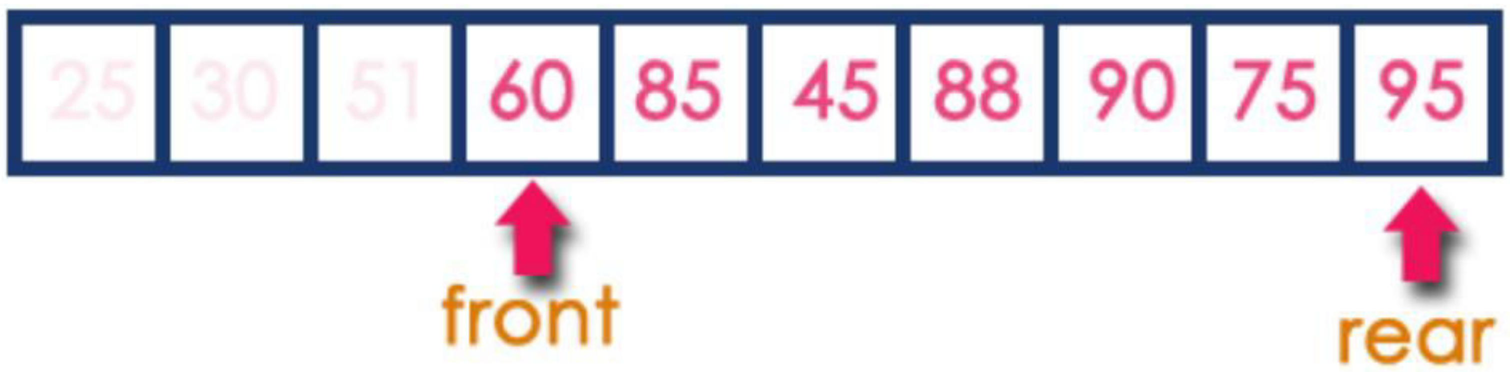
- insertion will denied even if room is available at the front.

Solution

- Use circular queue

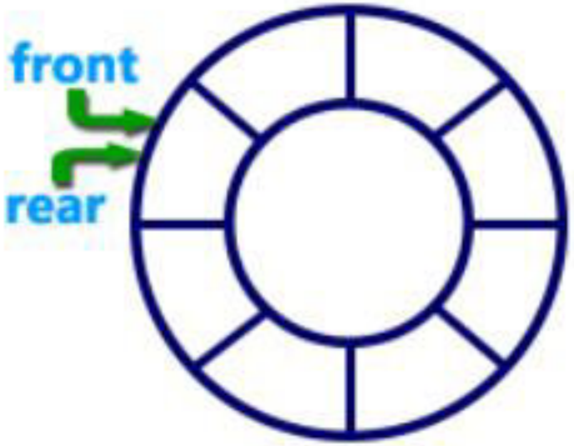
Queue is Full

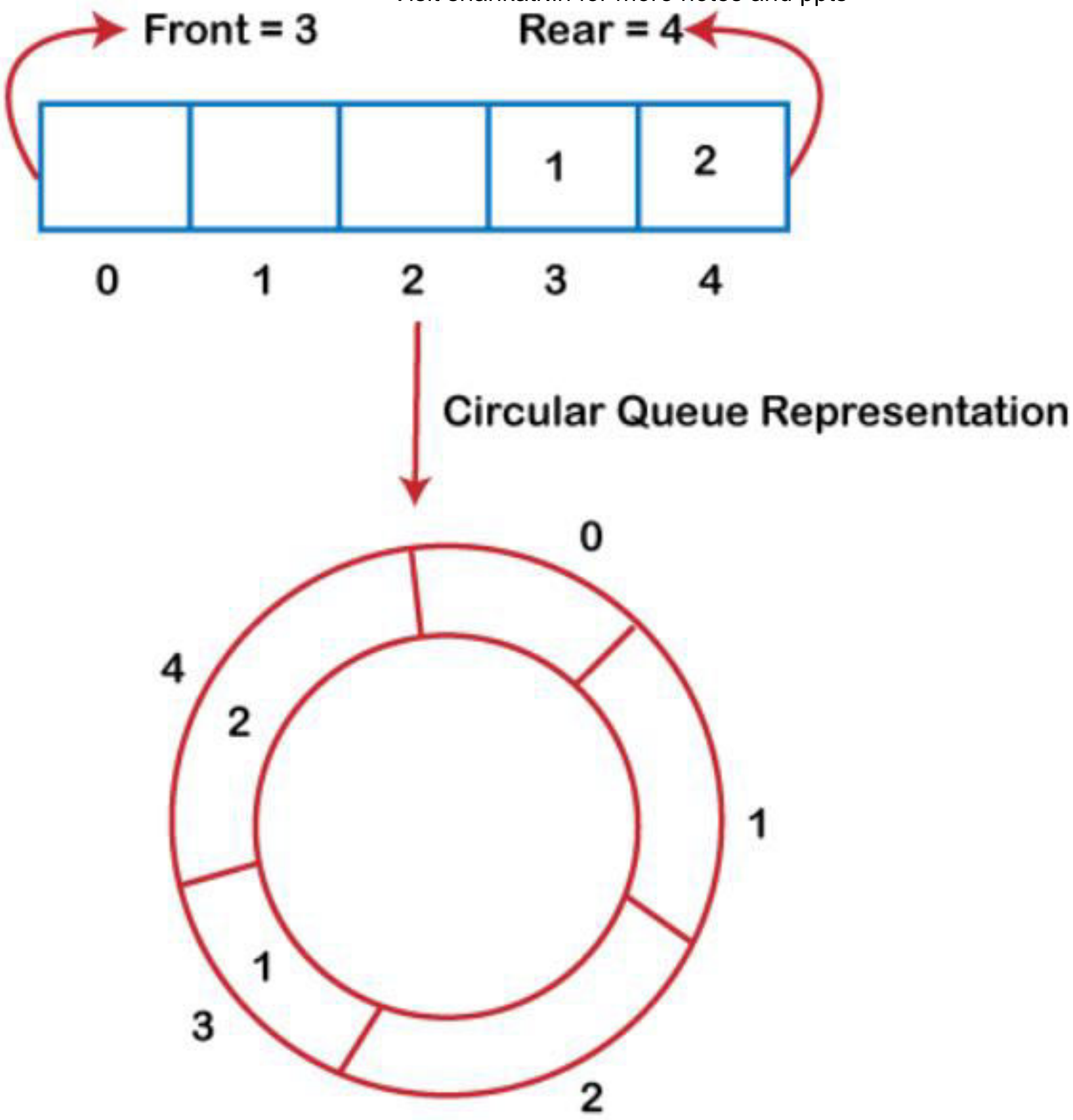
Queue is Full (Even three elements are deleted)



A 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.

It is also known as a Ring Buffer.




Front = 3
Rear = 4
1
2
0
1
2
3
4
Circular Queue Representation
0
1
2
3
4
2
1

Circular Queue

Physically circular array same as ordinary array, say Q[0..N-1],

- but Q[0] comes in between Q[1] and Q[N-1] .

Both pointers will move in same direction.

Example:

- If the current pointer is at position i, then the next location will be
  (i+1) MOD SIZE.
- i.e, if size =8
- Current position = 1 then next position = 2
- Current position = 7 then next position = (7+1) MOD 8 =0

States of the Circular Queue

Circular Queue is empty

- FRONT=-1
- REAR=-1

Circular Queue is full

- FRONT=(REAR+1) MOD SIZE
- i.e 0=(3+1) mod 4 → Queue FULL

Insertion(C-ENQUEUE)

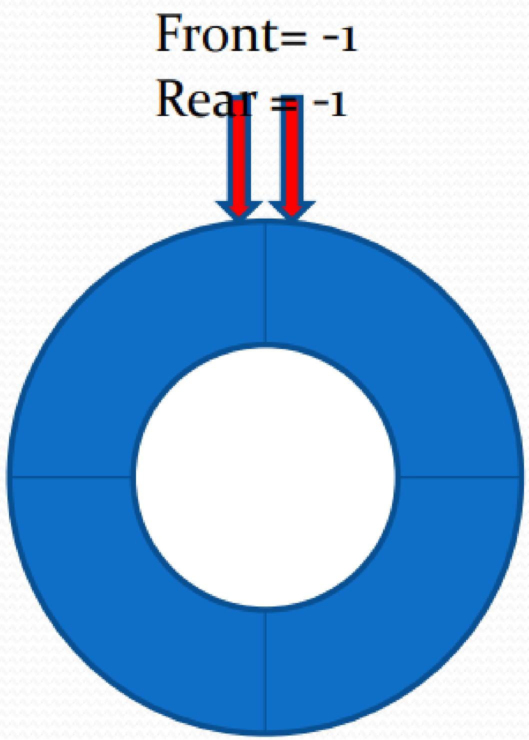
Initially the queue will be initialized as front= -1 and rear = -1

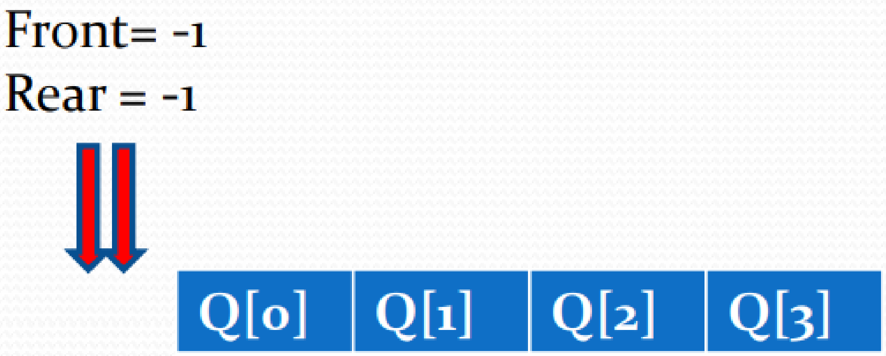
Before inserting check whether the queue is full or not.

If not full, then insert the element to (REAR+1) MOD SIZE

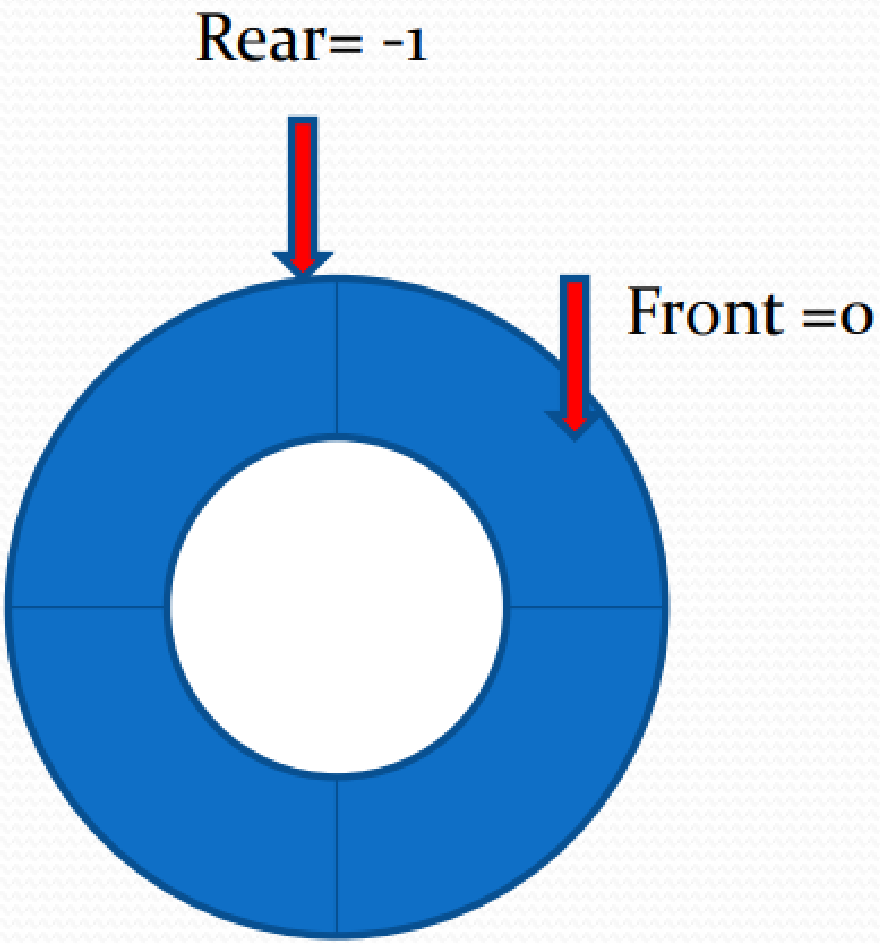
Make sure that the front always points to the first element by incrementing the front pointer when the first element is inserted.

- Initially

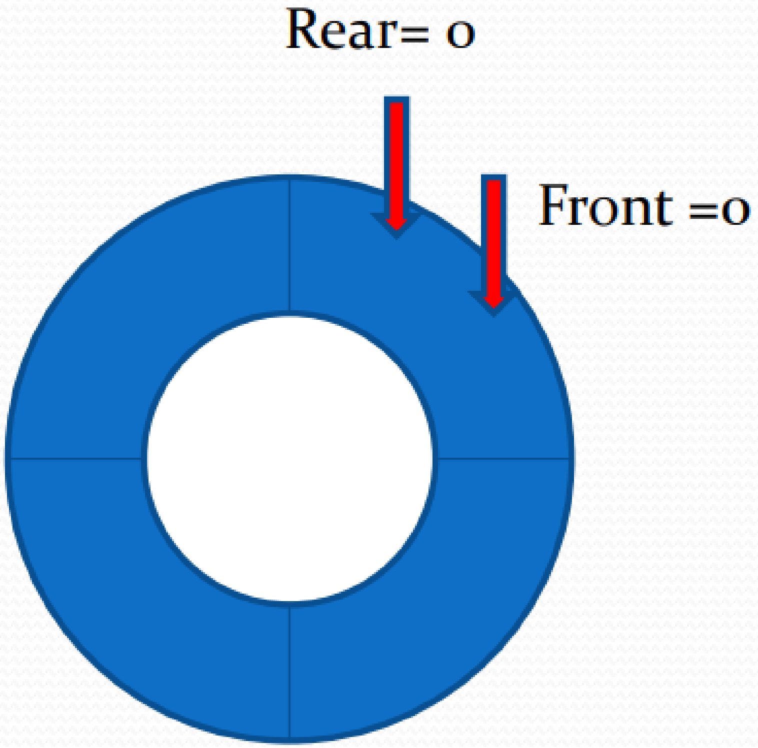




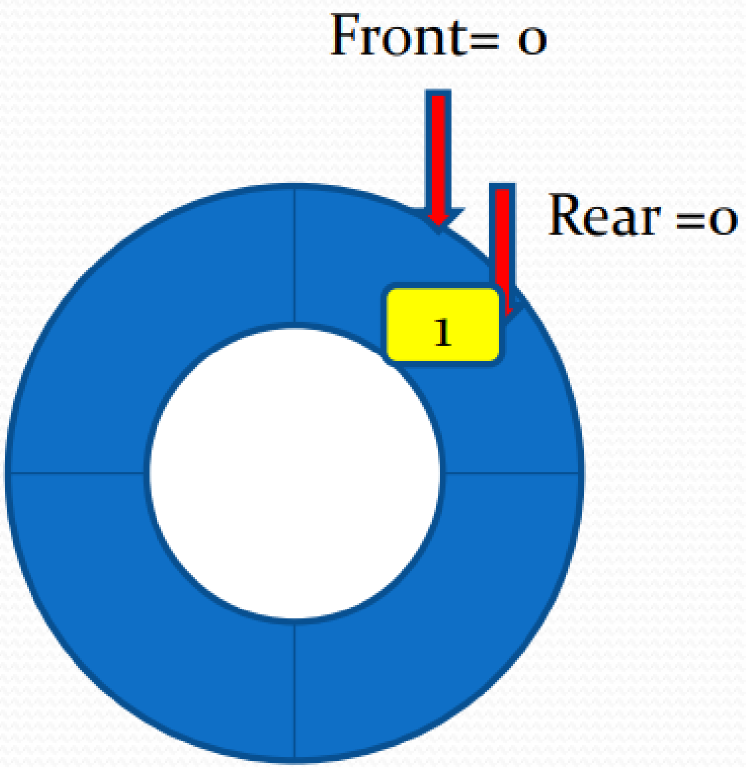
- C-Enqueue (1)



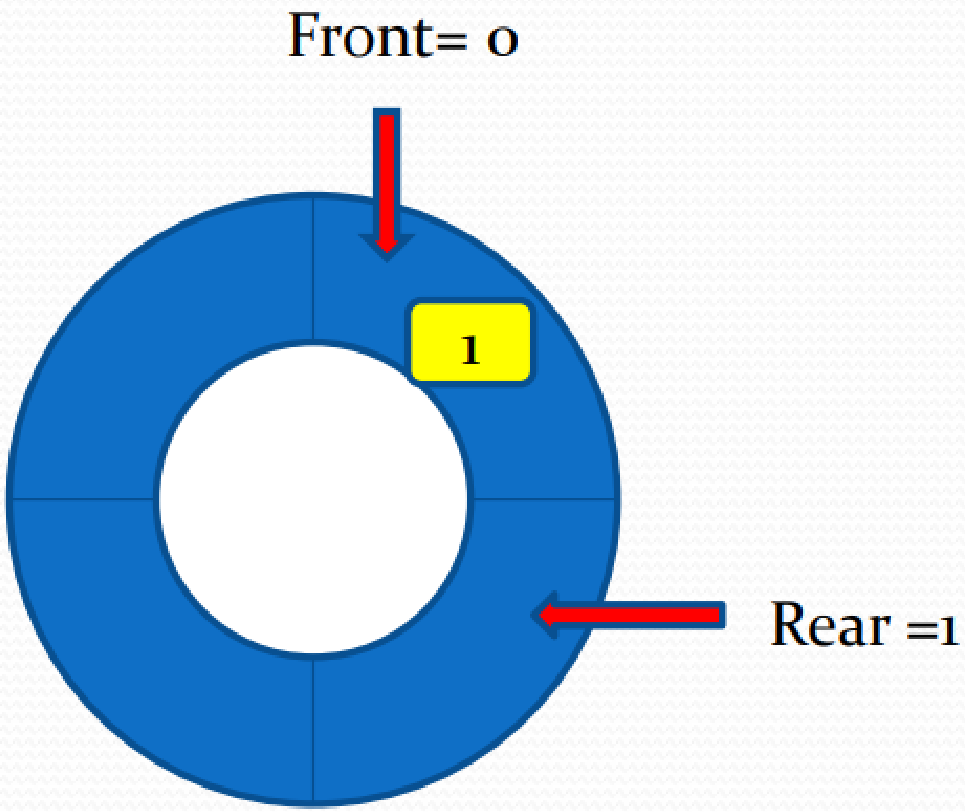
- C-Enqueue (1)



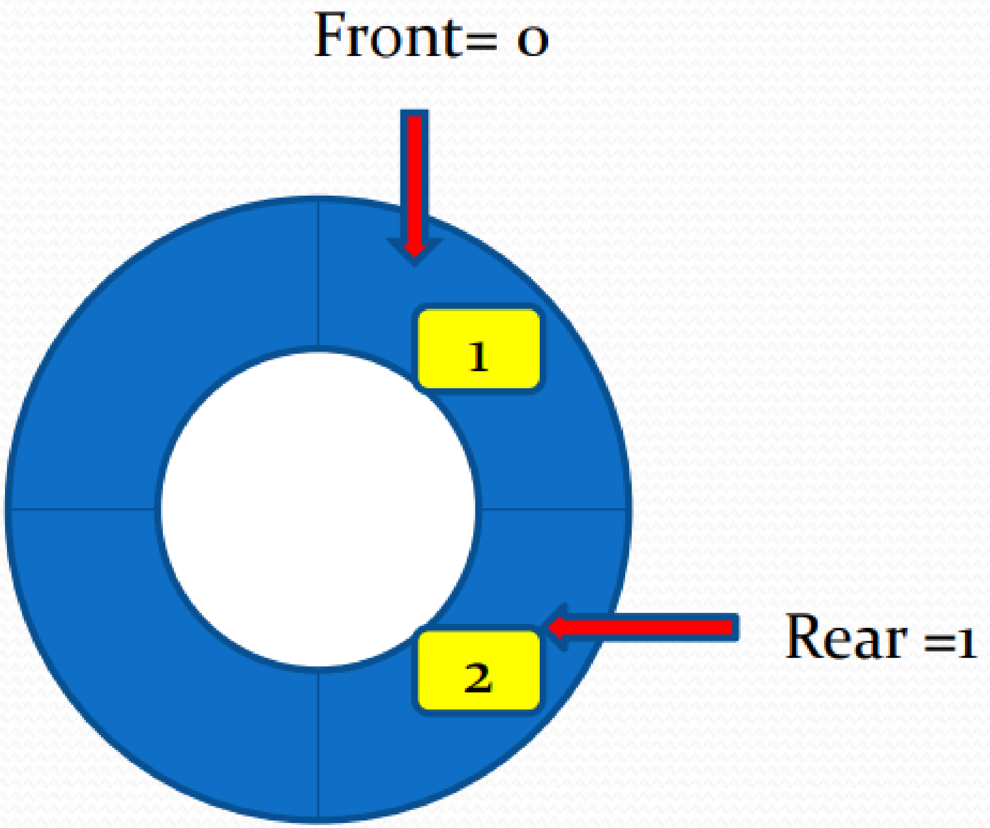
- C-Enqueue (1)



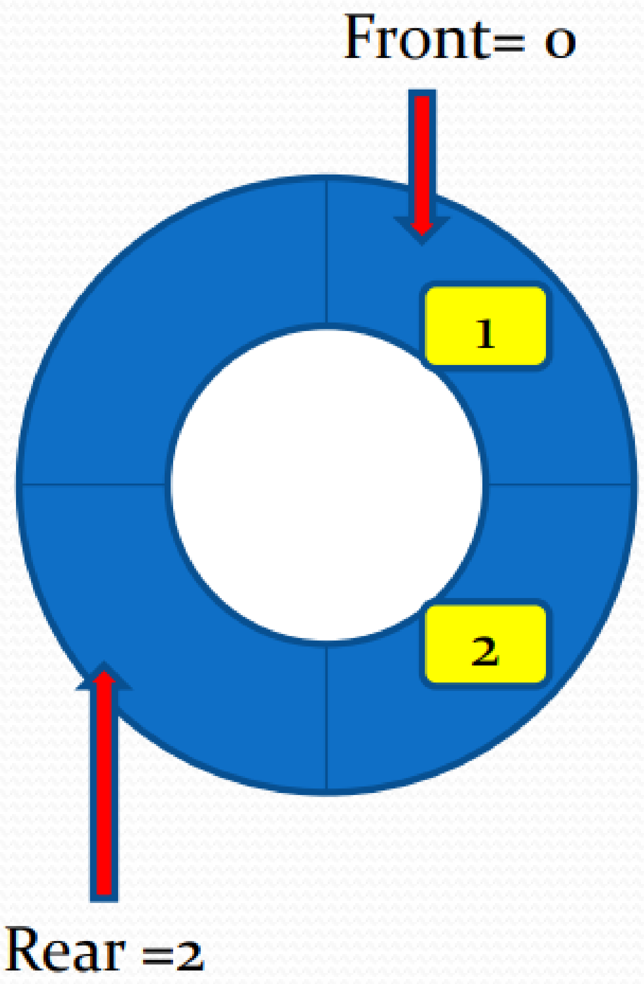
- C-Enqueue (2)



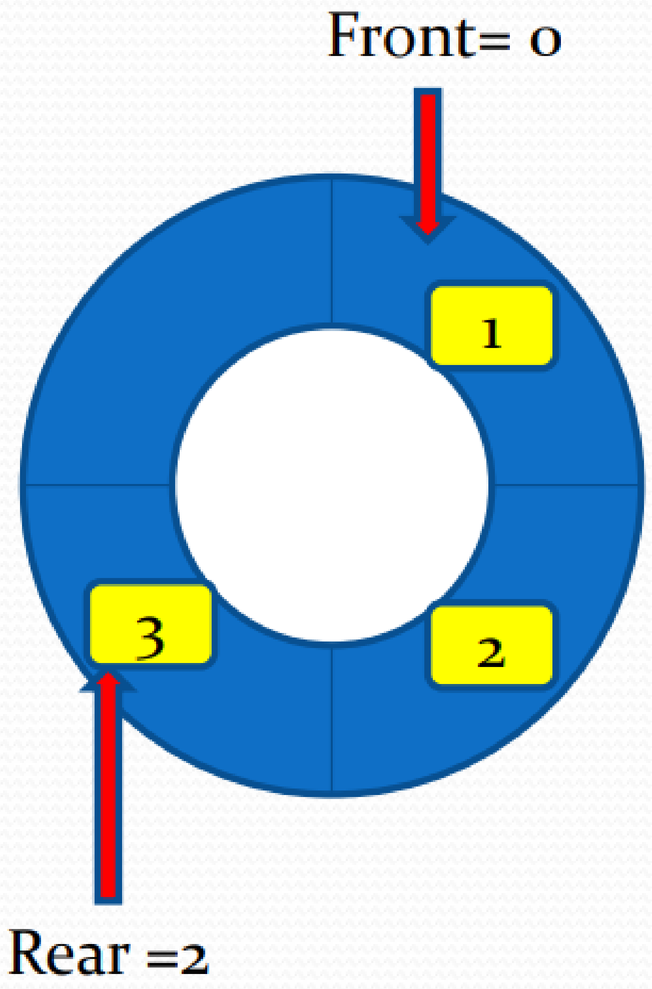
- C-Enqueue (2)



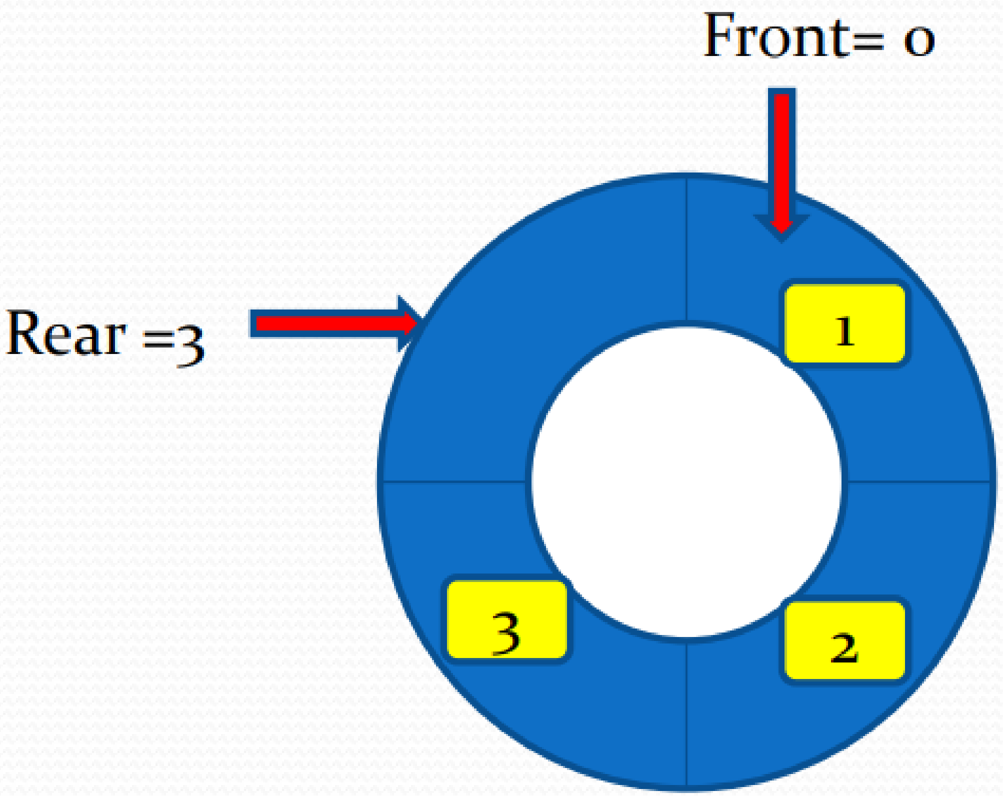
- C-Enqueue (3)



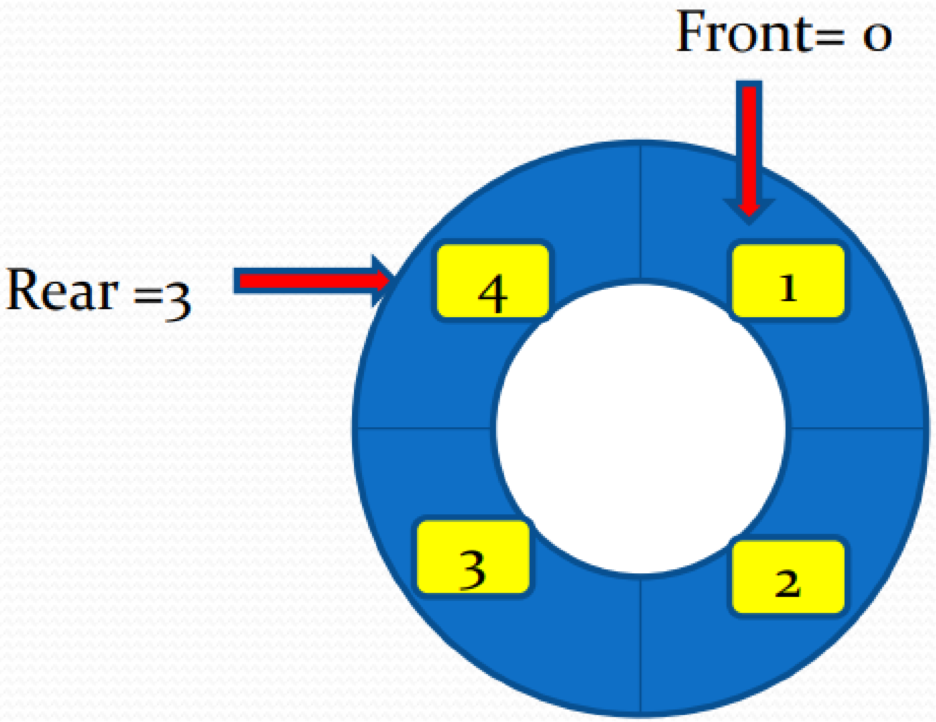
- C-Enqueue (3)



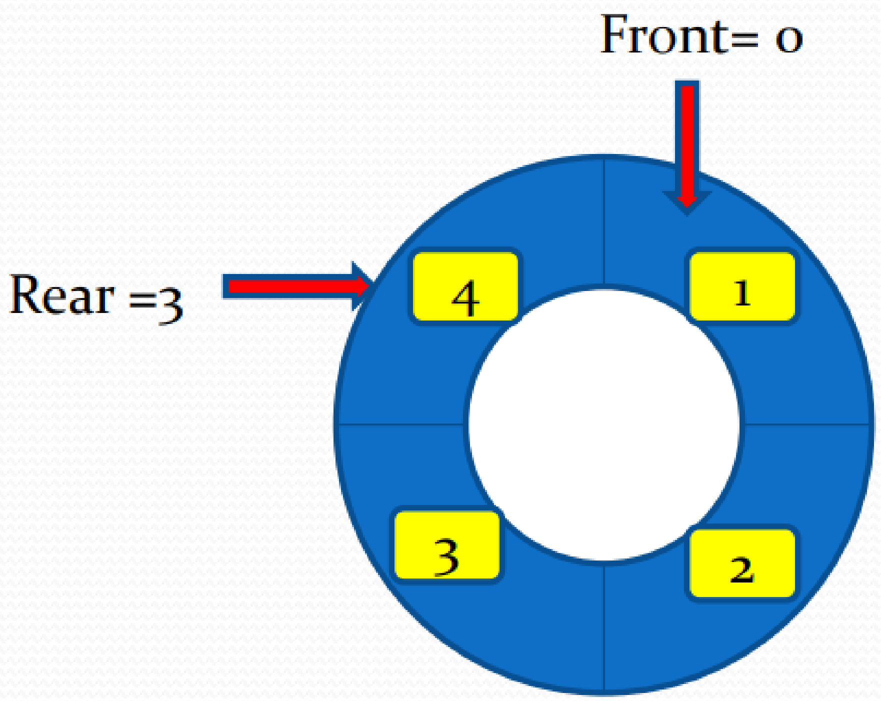
- C-Enqueue (4)



- C-Enqueue (4)



- C-Enqueue (5)



next = (rear +1) mod size
=(3+1) mod 4)
=0

But next =front

→ Queue is full, insertion not possible

Algorithm C-ENQUEUE(ITEM)

Input: An element ITEM to be inserted into the circular queue

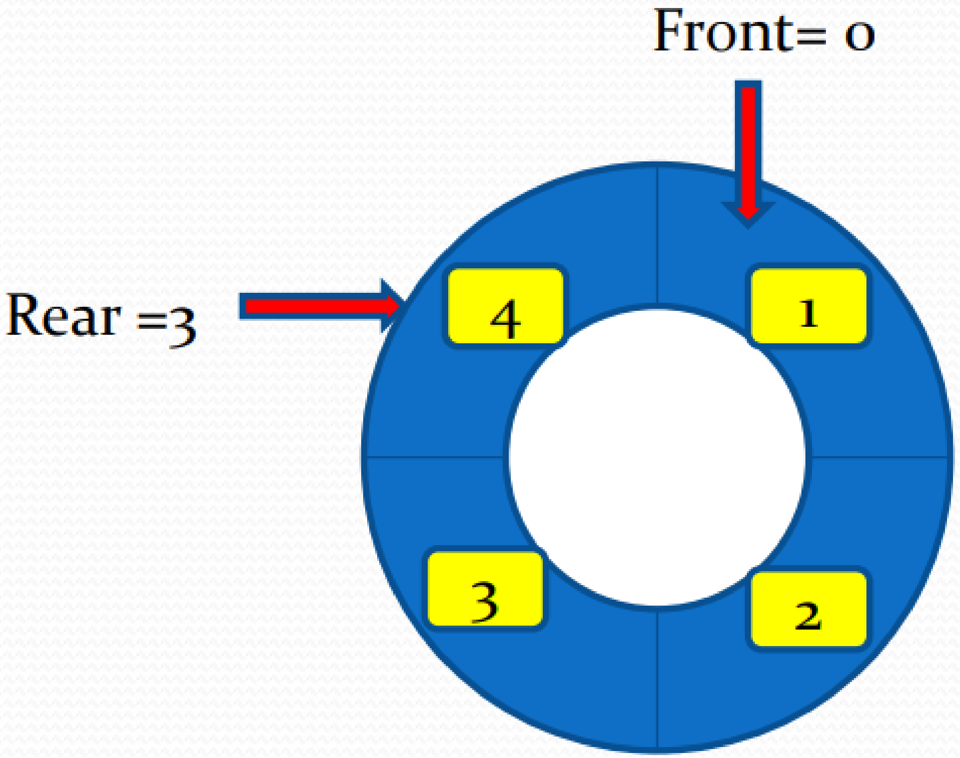
Output: Circular queue with the ITEM , if not full

Data Structure: CQ is implemented by using array

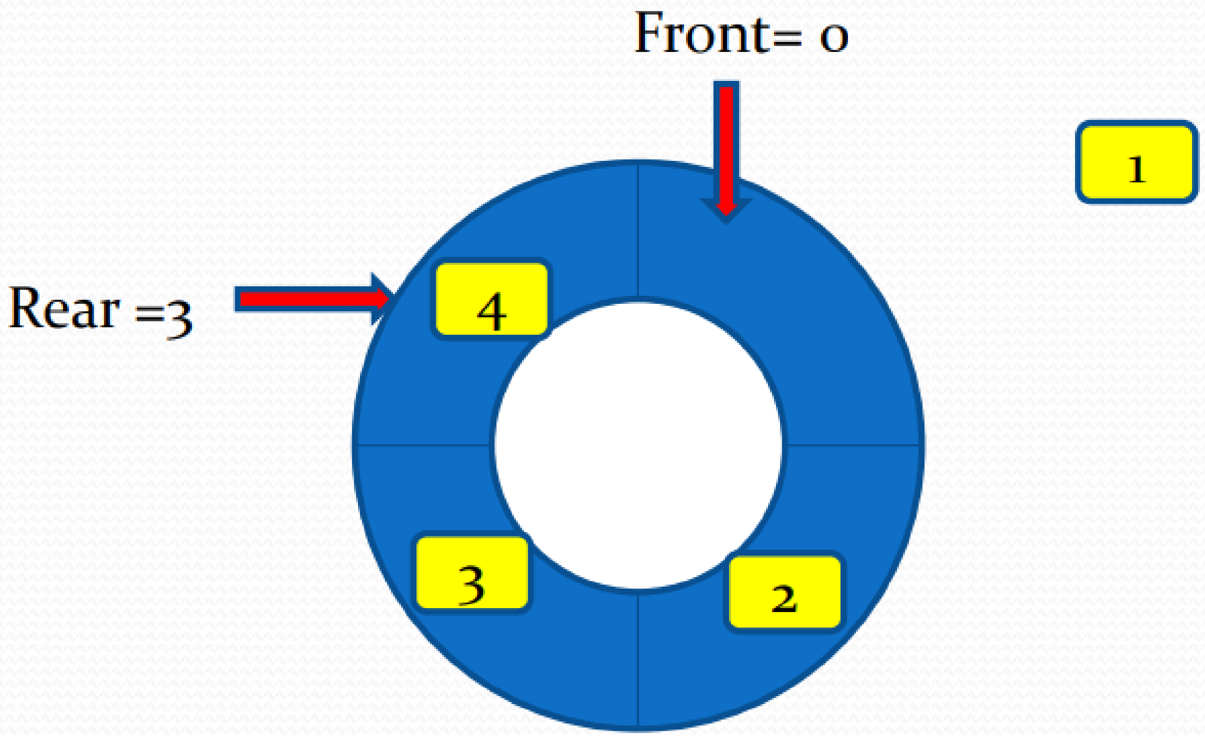
Steps:

1. next=(REAR+1) ***mod*** SIZE
2. If next=FRONT
    1. Print" Queue is full"
3. Else
    1. REAR=next
    2. Q[REAR]=ITEM
    3. If FRONT=-1 then
        1. FRONT=0
    4. EndIf
4. EndIf
5. Stop

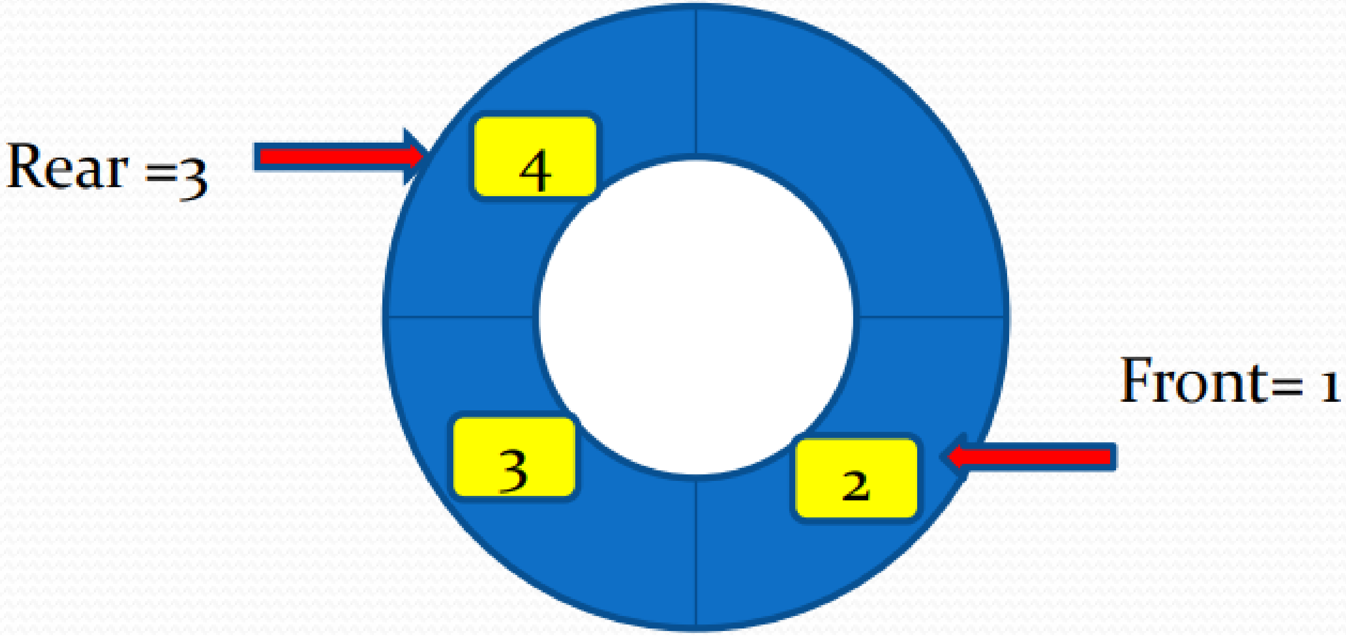
- Delete an element from FRONT of the queue



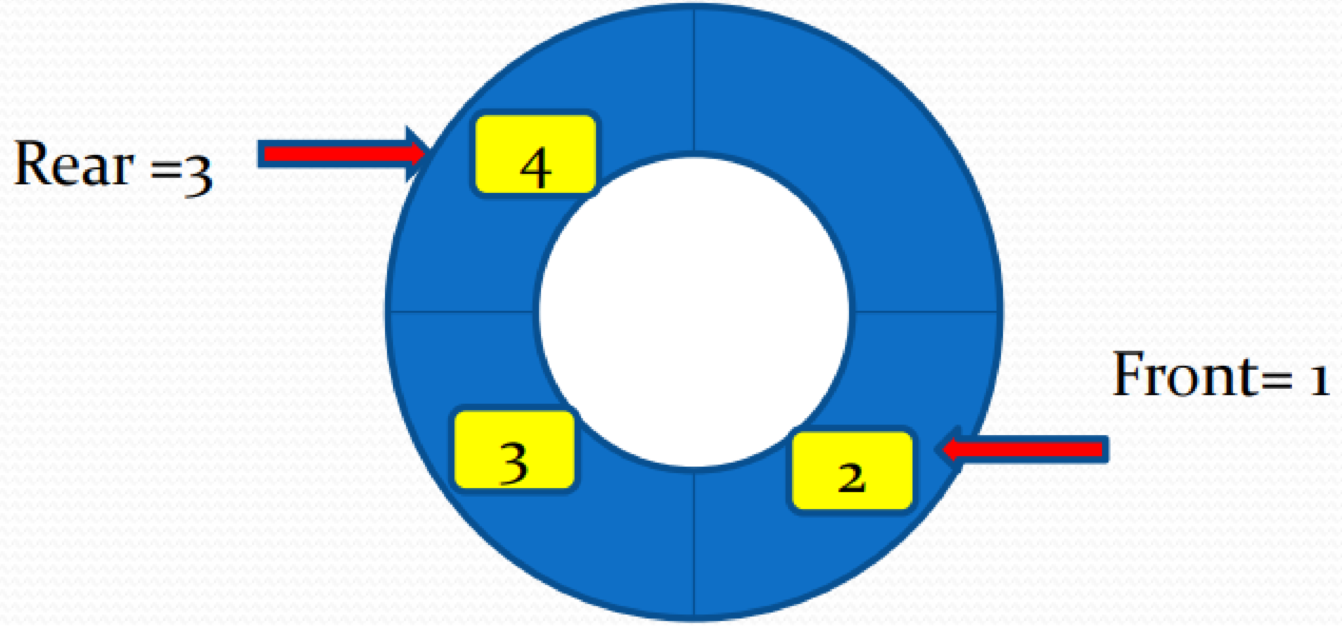
- Delete an element from FRONT of the queue



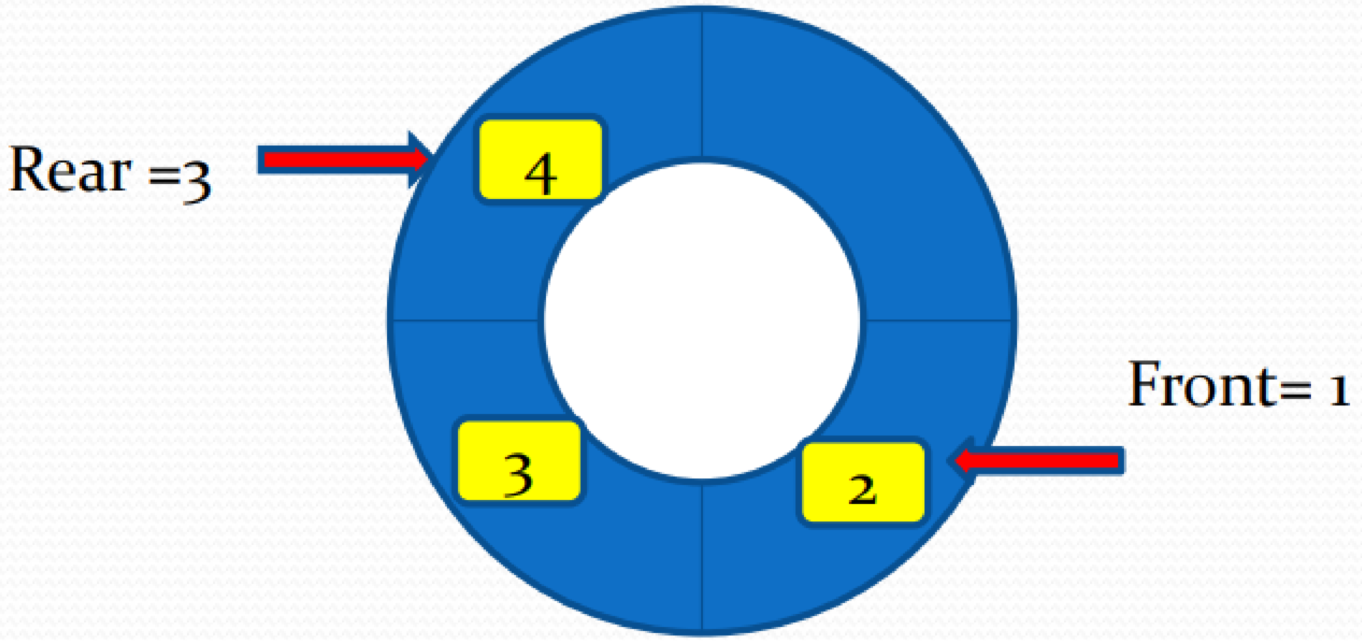
- Deleted an element



Is it possible to add another element into this empty space ??



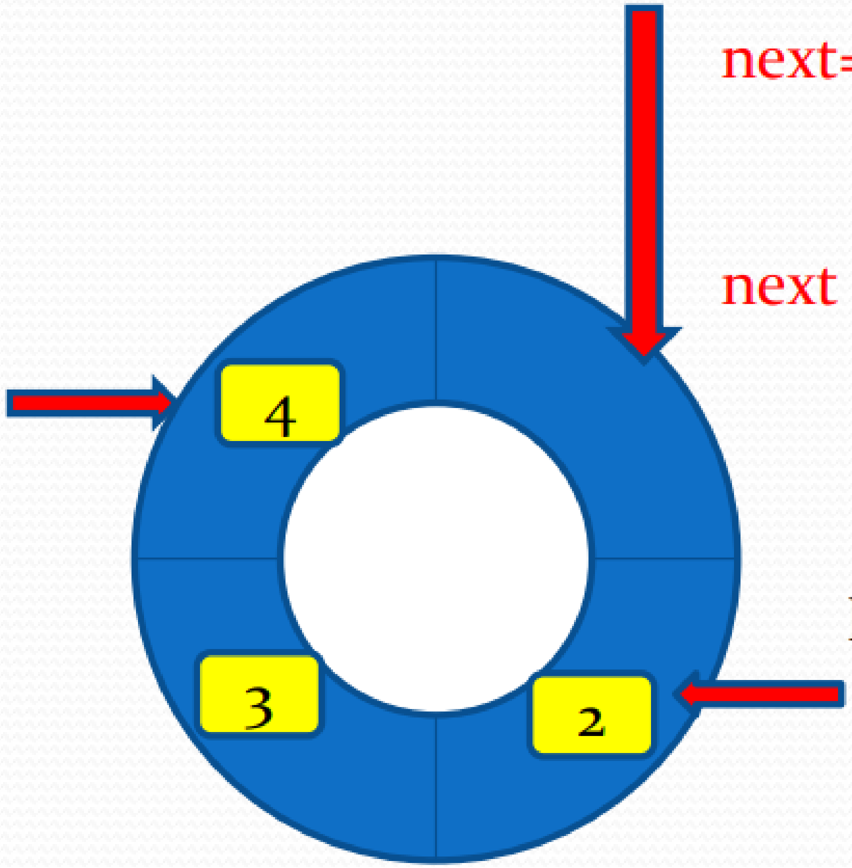
- Insert (5)



- Insert (5)

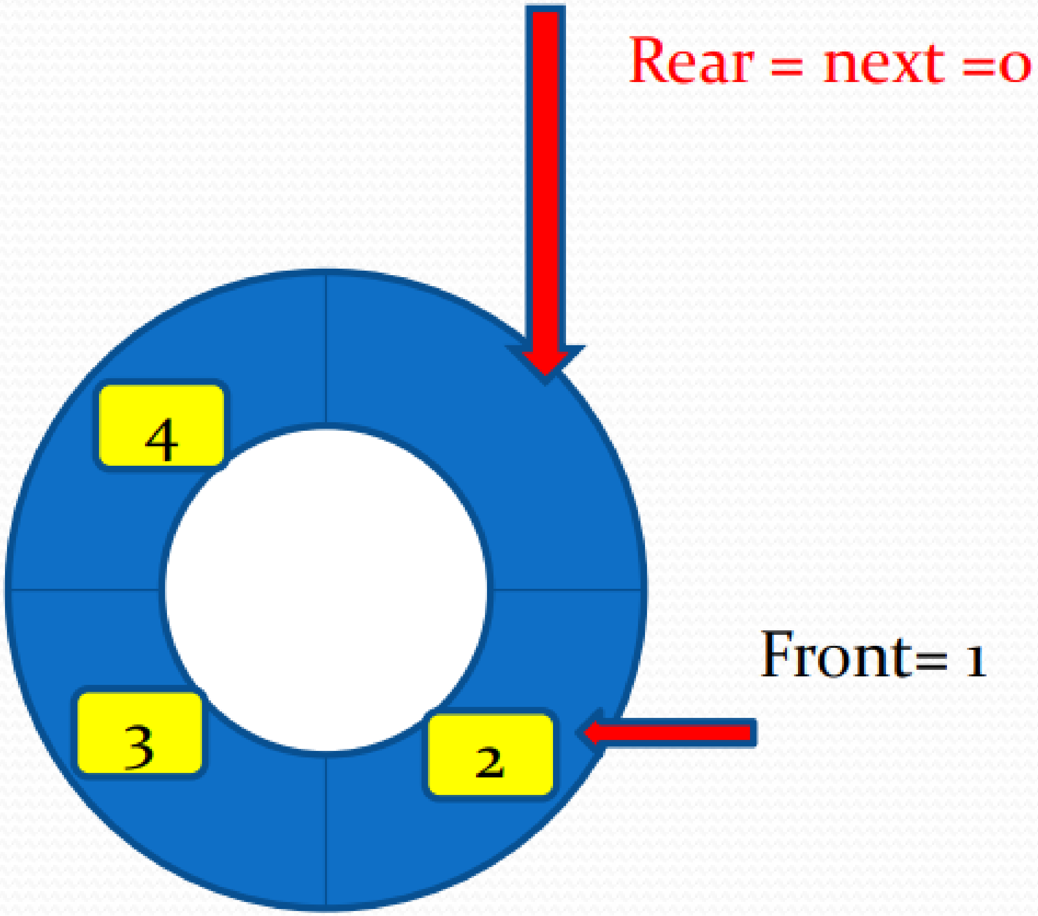
next= (rear+1) mod size
=(3+1) mod 4)
=0
next != Front → queue not full

Rear =3

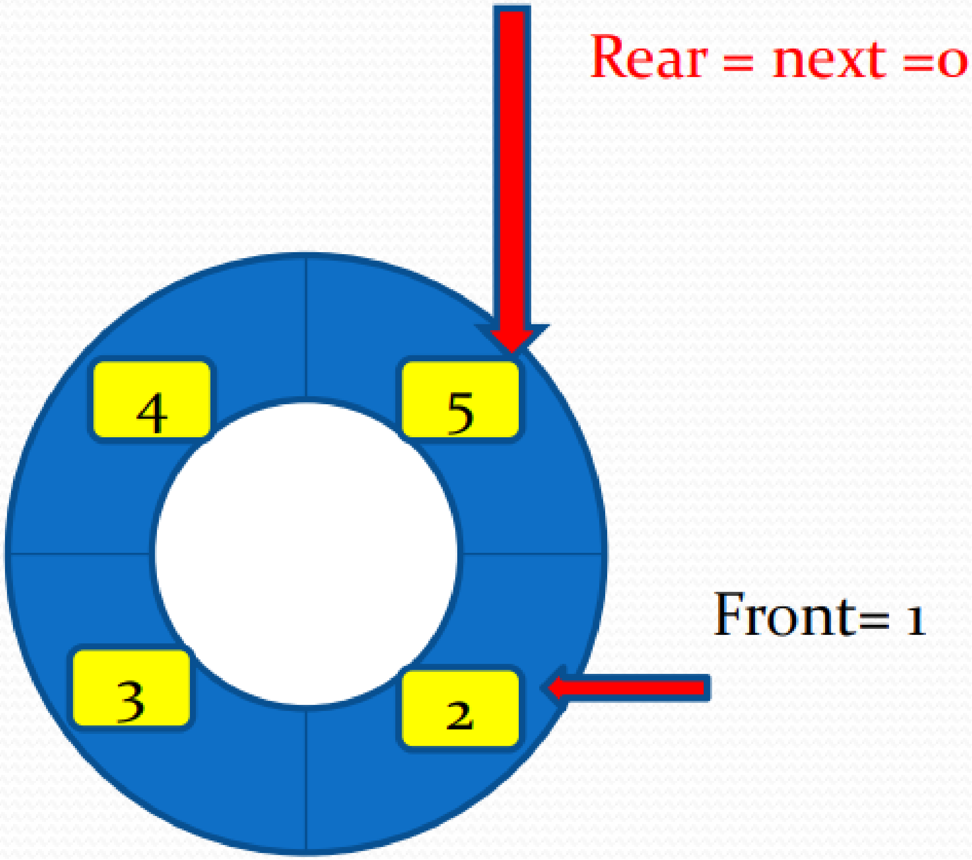


Front= 1

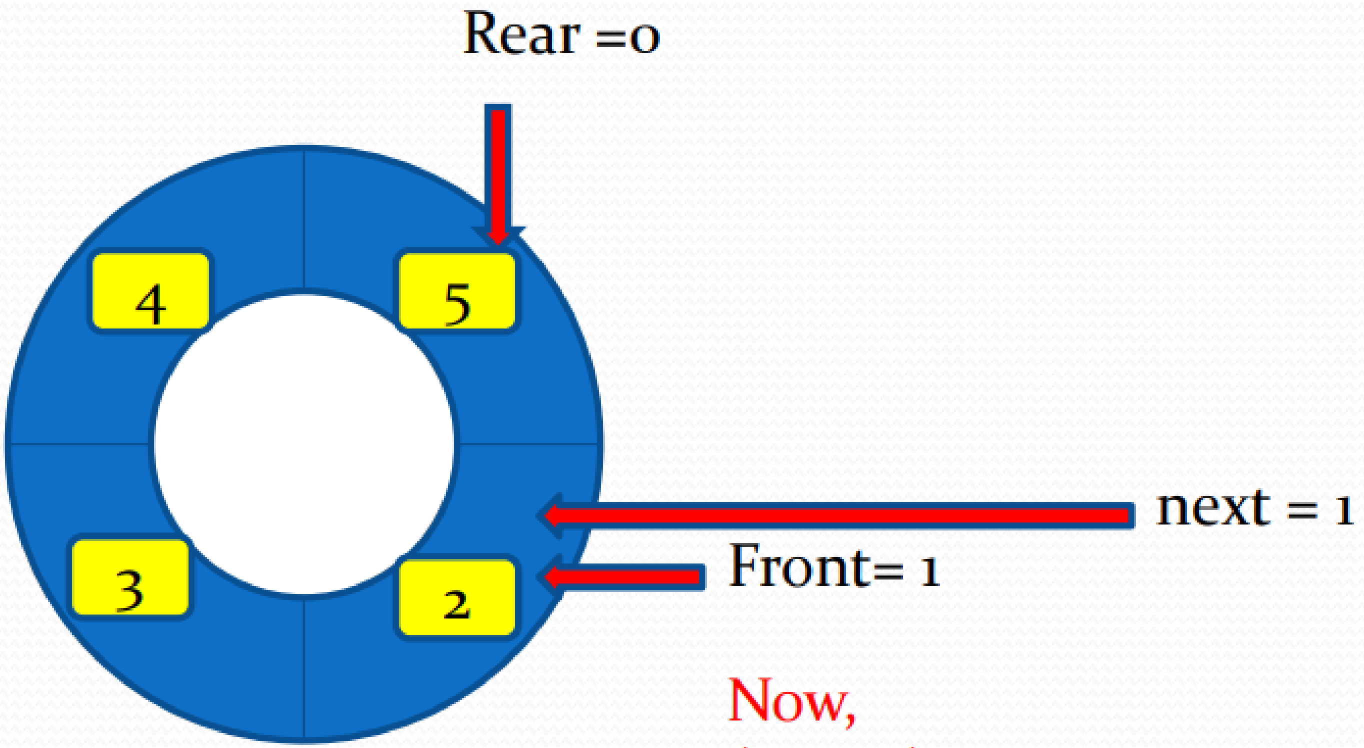
- Insert (5)



- Insert (5)



- Insert (6)



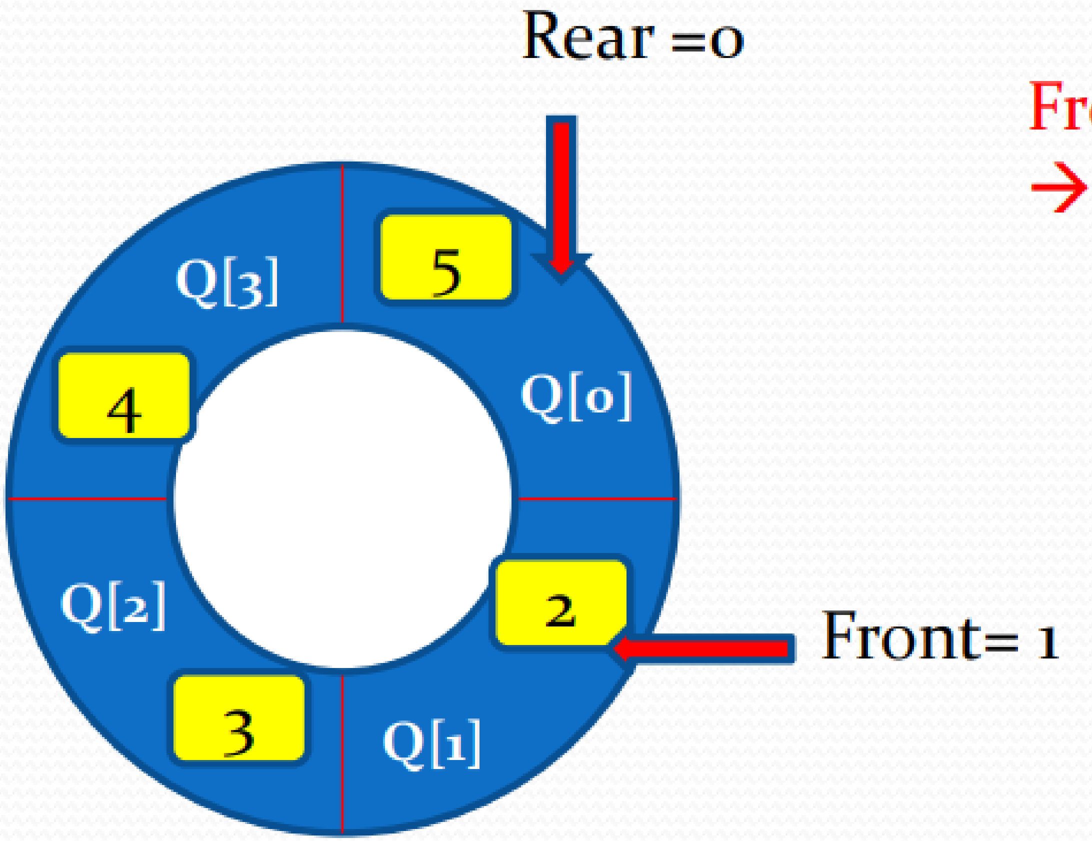
Now,
(rear+1) mod size= front
→Queue is full
→Insertion impossible

Deletion(C- DEQUEUE)

Before deleting, check whether the circular queue is empty or not.

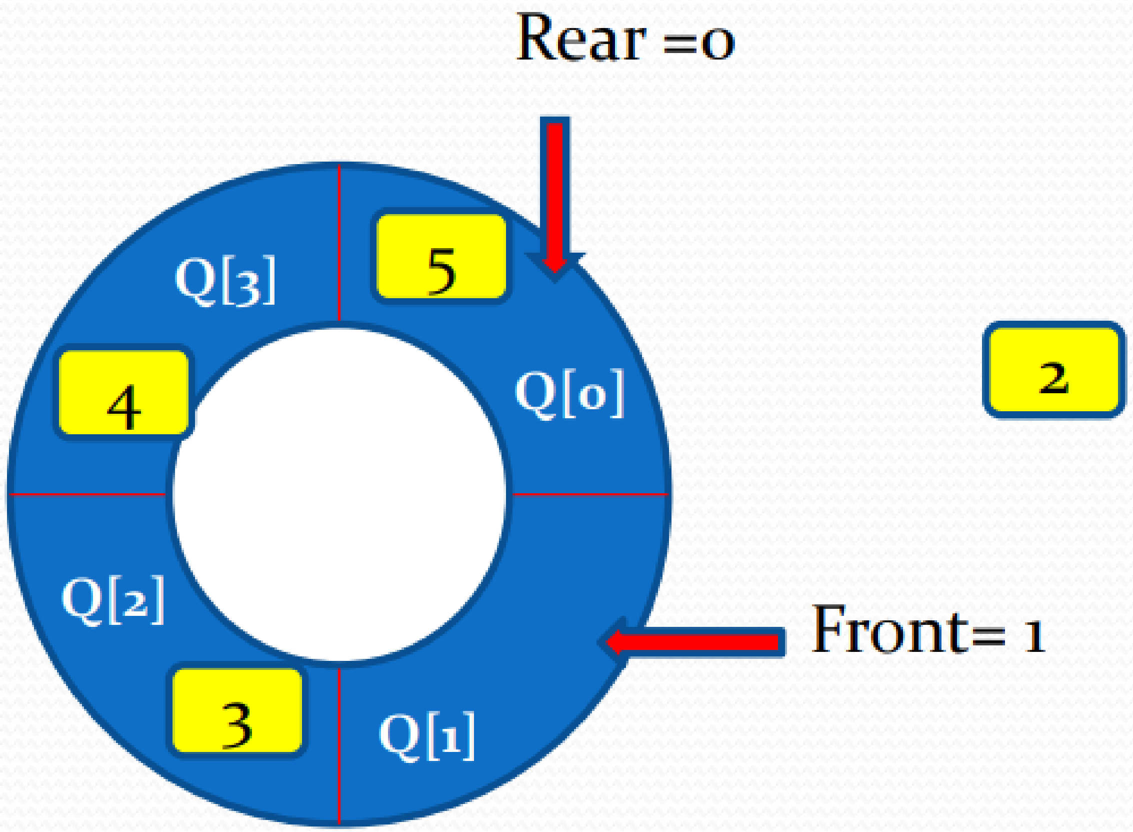
If not empty, then delete the element

Make sure that the FRONT and REAR always points to -1 when the last element is deleted.

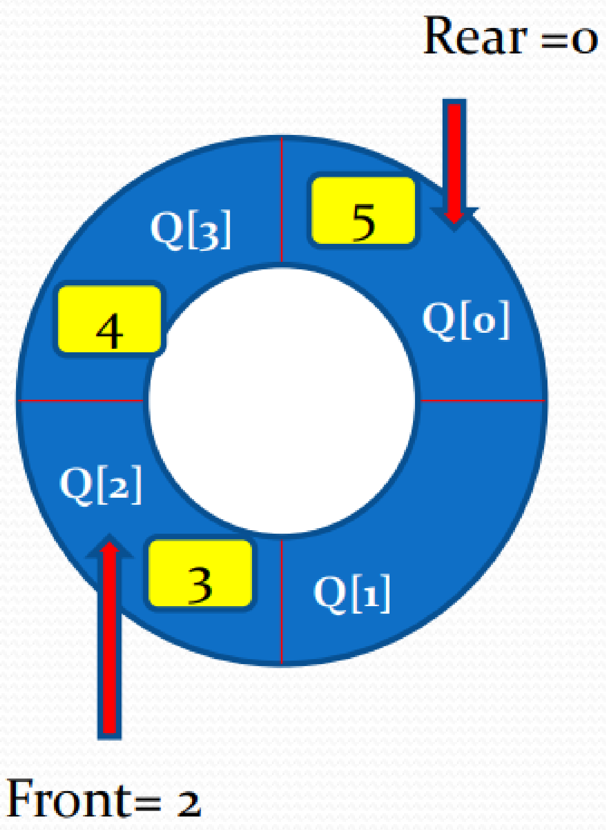


Front != -1
→ CQ not empty

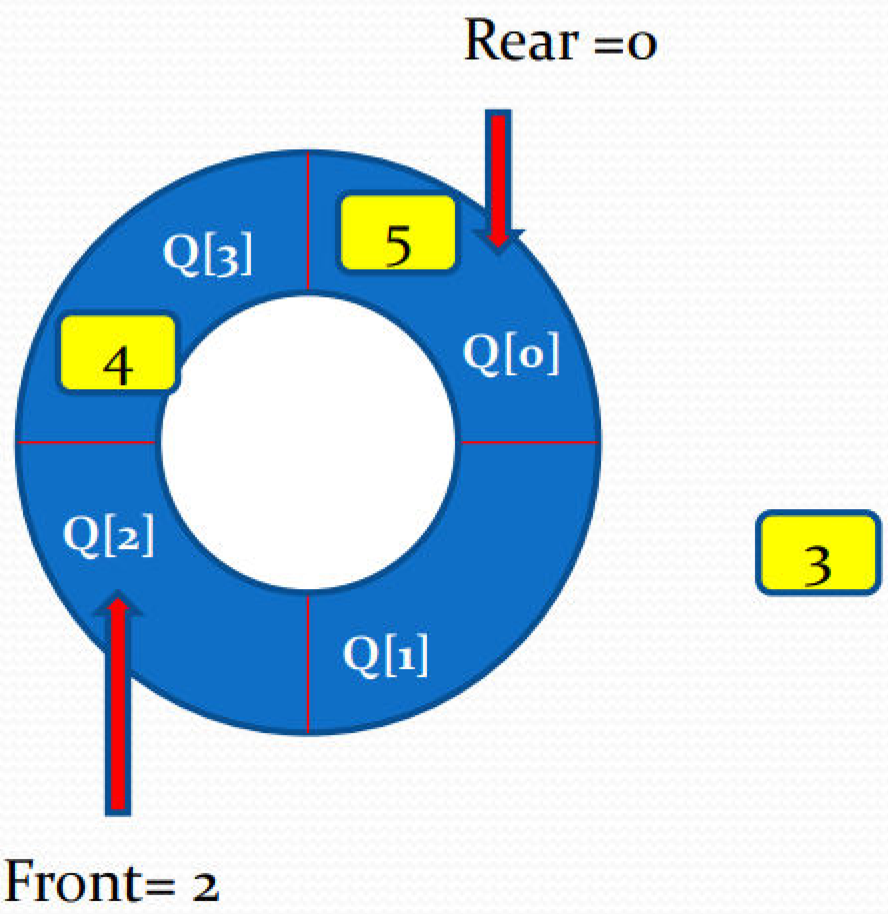
- C-dequeue



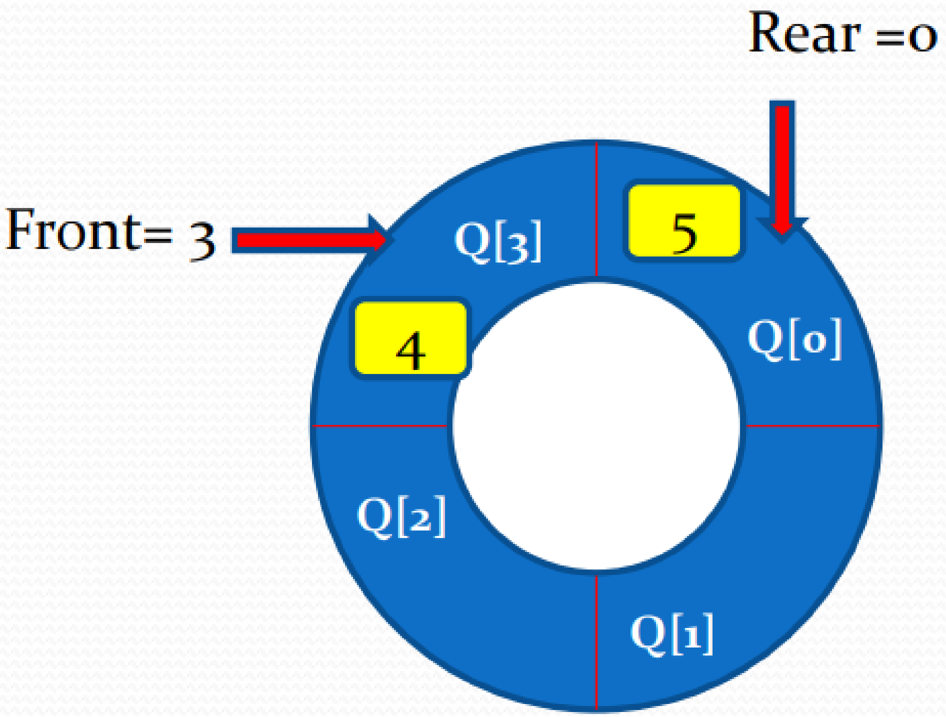
Deletion(C- DEQUEUE)



- C-dequeue

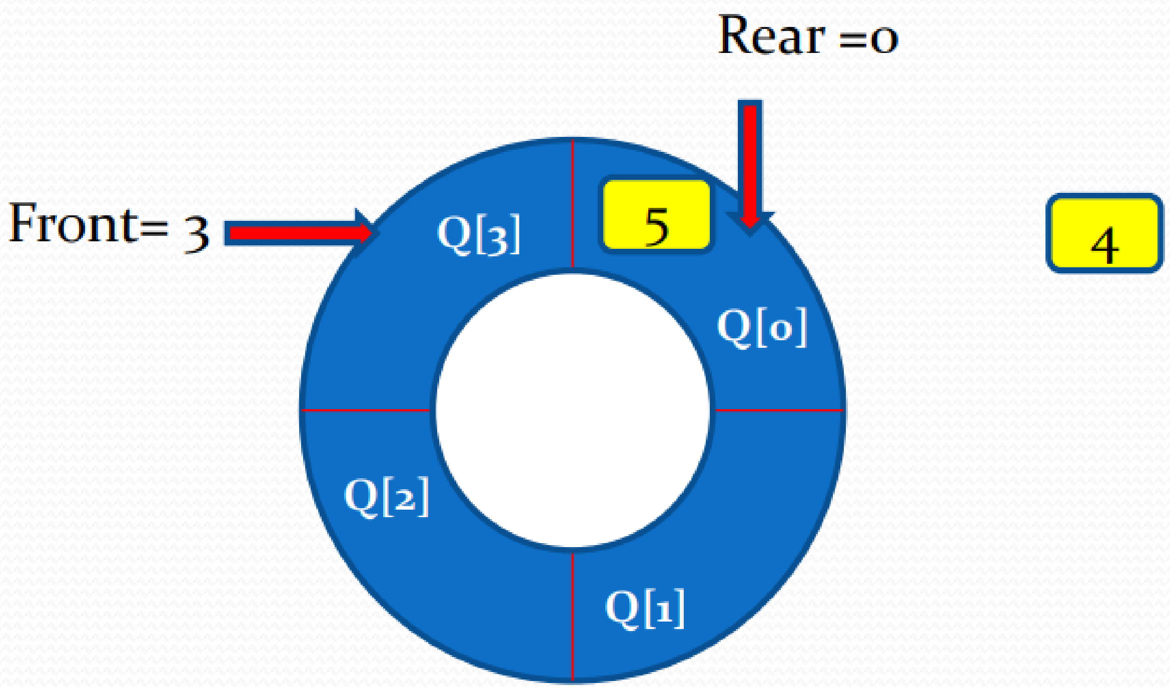


Deletion(C- DEQUEUE)

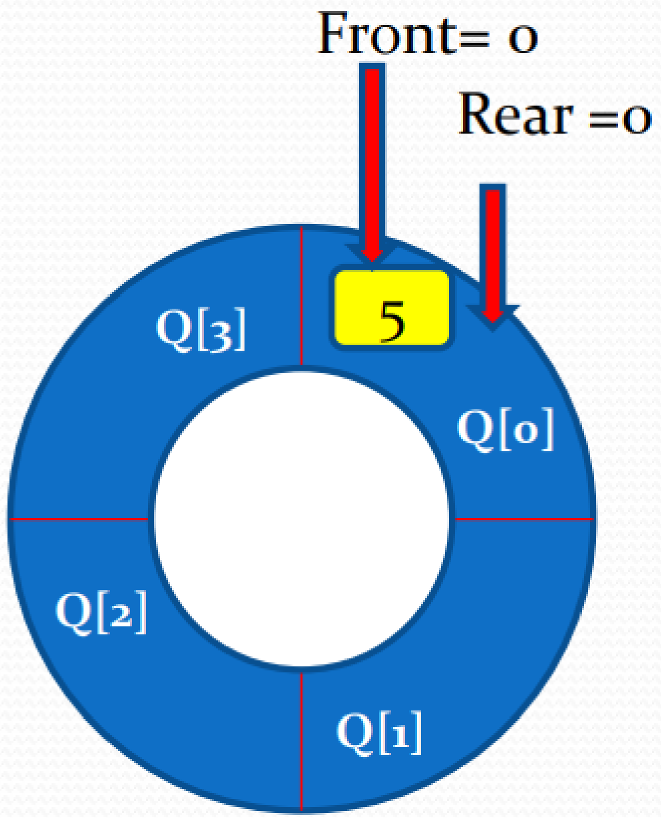


Deletion(C- DEQUEUE)

- C-dequeue



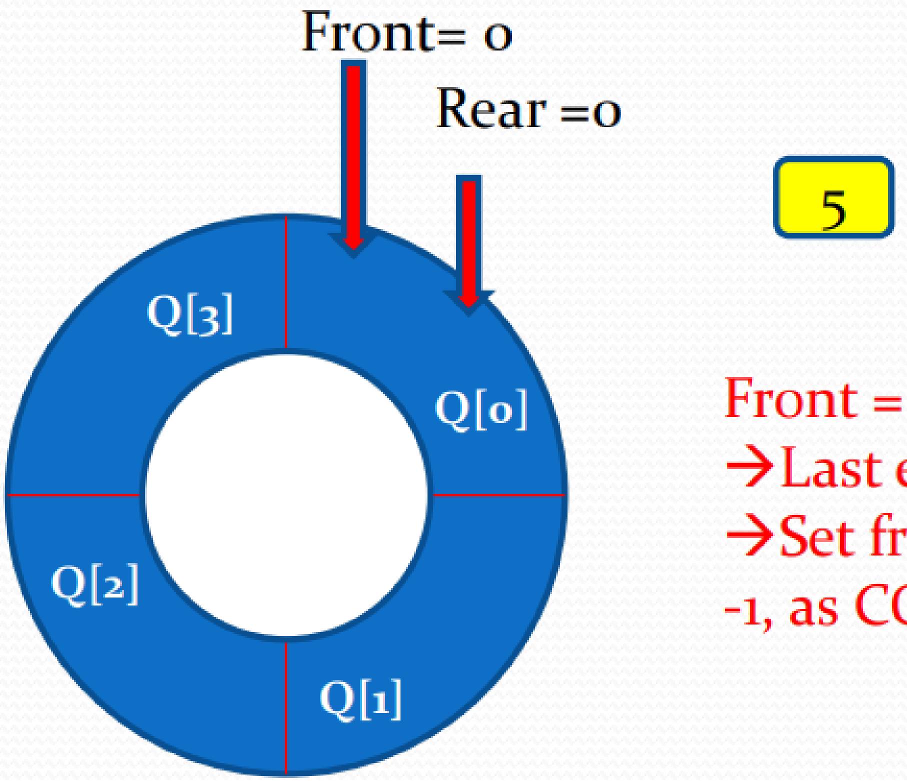
Deletion(C- DEQUEUE)



Front = rear
→Last element
→Set front and rear to -1, as CQ is empty

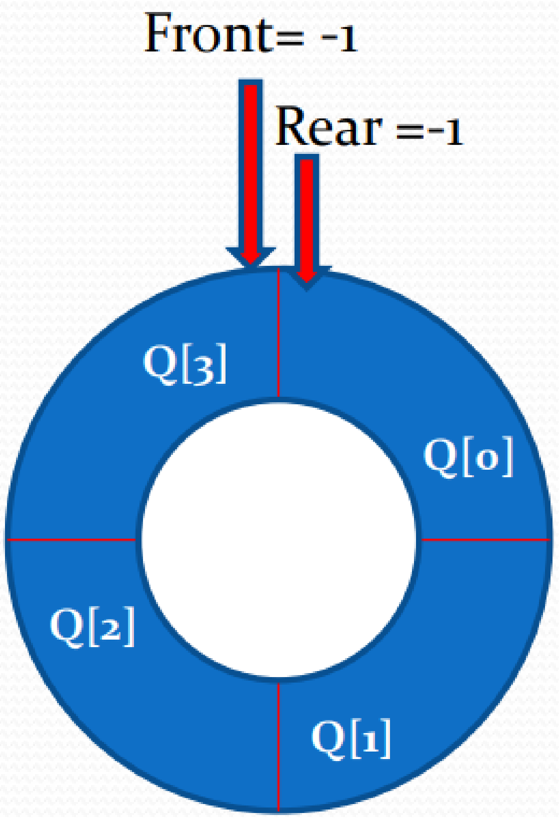
Deletion(C- DEQUEUE)

- C-dequeue



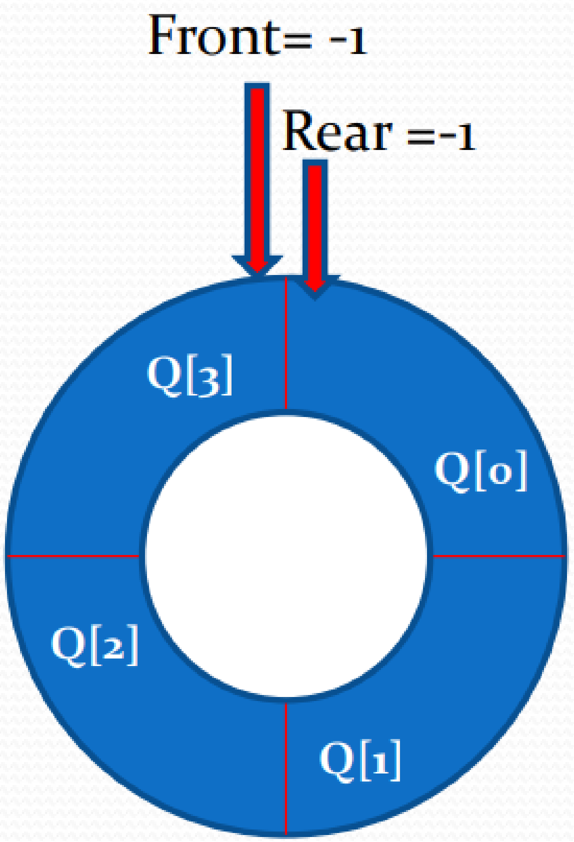
Front = rear
→Last element
→Set front and rear to -1, as CQ is empty

Deletion(C- DEQUEUE)



Deletion(C- DEQUEUE)

- C-dequeue



CQ empty
Deletion impossible

Algorithm DEQUEUE()

Input: A Queue with n elements

Output: The deleted element is ITEM if the Queue is not empty.

Data Structure: CQ is implemented using array.

Steps:

1. If FRONT=-1 then
   1. Print "Queue is Empty"
2. Else
   1. ITEM=Q[FRONT]
   2. If FRONT=REAR
      1. FRONT=-1
      2. REAR=-1
   3. Else
      1. FRONT=(FRONT+1) mod SIZE
   4. EndIf
3. EndIf
4. Stop

Activity

Consider a circular queue of size=4. Do following operations

1. ENCQUEUE (A)
2. ENCQUEUE (B)
3. ENCQUEUE (C)
4. ENCQUEUE (D)
5. DECQUEUE
6. ENCQUEUE (E)
7. DECQUEUE
8. ENCQUEUE (F)
9. DECQUEUE
10. DECQUEUE
11. DECQUEUE
12. DECQUEUE

Assume that initially the queue is empty, that is, $FRONT = REAR = 0$.

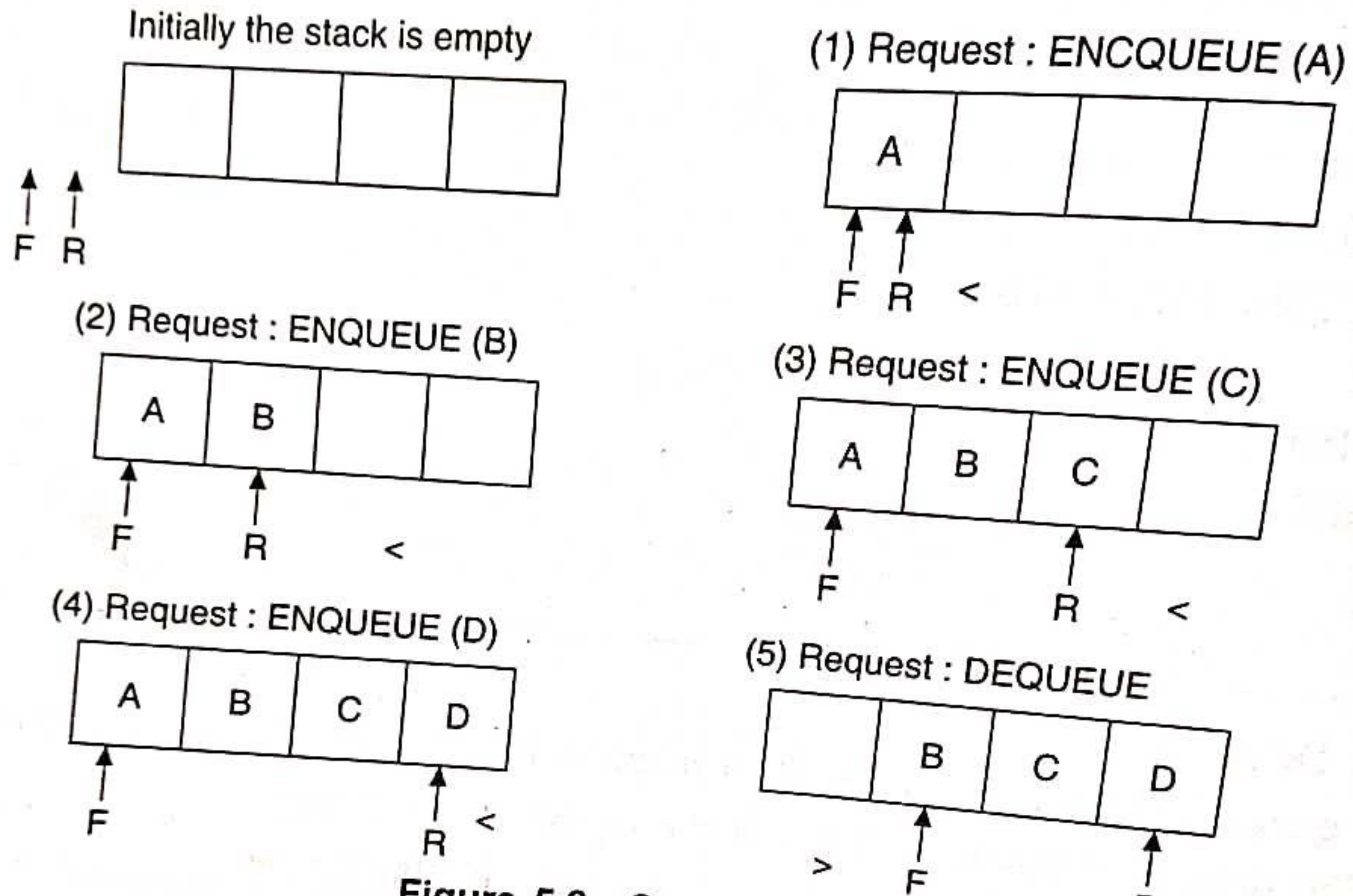


Figure 5.8 Continued

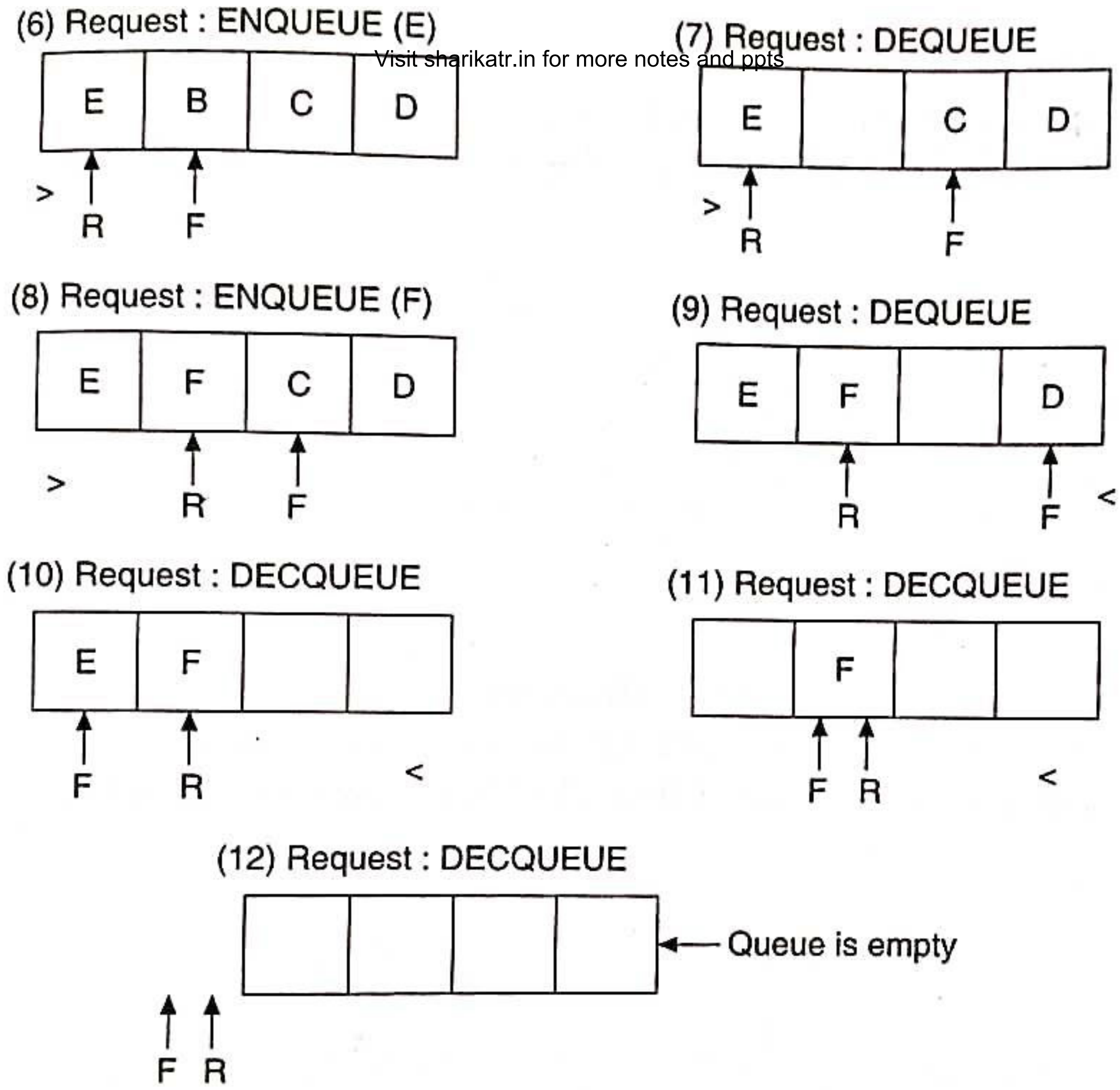


Figure 5.8 Tracing insertion and deletion operations on a circular queue.

Double Ended Queue Deque

Also known as a double-ended queue

An ordered collection of items similar to the queue.

Insertion and deletion possible at both ends.

This hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.

It does not require the LIFO and FIFO orderings that are enforced by stack and queue

Representation of deque

Deque operations

addFront(item) or Push_DQ(ITEM) adds a new item to the front of the deque.

addRear(item) or Inject(ITEM) adds a new item to the rear of the deque.

removeFront() or Pop_DQ() removes the front item from the deque

removeRear() or Eject() removes the rear item from the deque.

addFront(int item)

Steps:

1. if(front==0 and rear=n-1)
   1. Print " Queue is full.."
   2. Exit
2. If(front=-1)
   1. Rear=0
   2. Front=0
   3. Q[front]=item
3. Else if(front>0)
   1. front=front-1
   2. Q[front]=item
4. Else
   1. i=rear
   2. While(i>=front)
      1. Q[i+1]=Q[i]
      2. i=i-1
   3. End while
   4. Q[front]=item
   5. Rear++
5. End if

1	**2**	**3**	

F=0 R=2

i=2(rear)
Q[i+1]=Q[i]// shifting one position till i become 0 results to

4	**1**	**2**	**3**

F=0 R=3

addRear(int item)

Steps:

1. if(front==0 and rear=n-1)
   1. Print " Queue is full.."
   2. Exit
2. If(front=-1)
   1. Rear=0
   2. Front=0
   3. Q[front]=item
3. Else if(rear<n-1)
   1. Rear++
   2. Q[rear]=item
4. Else
   1. i=front
   2. While(i<=rear)
      1. Q[i-1]=Q[i]
      2. i=i+1
   3. End while
   4. Q[rear]=item
   5. Front=front-1
5. End if

Int deleteFront()

Steps:

1. if(front==-1 and rear=-1)
   1. Print " Queue is empty.."
   2. Exit
2. Item=Q[front]
3. If(front==rear)
   1. Rear=-1
   2. Front=-1
4. Else
   1. Front++
5. End if
6. Return item

Int deleteRear()

Steps:

1. if(front==-1 and rear=-1)
   1. Print " Queue is empty.."
   2. Exit

2.Item=Q[rear]

3.if(front==rear)
   1. Rear=-1
   2. Front=-1

4.Else
   1. Rear--

5.End if

6.Return item

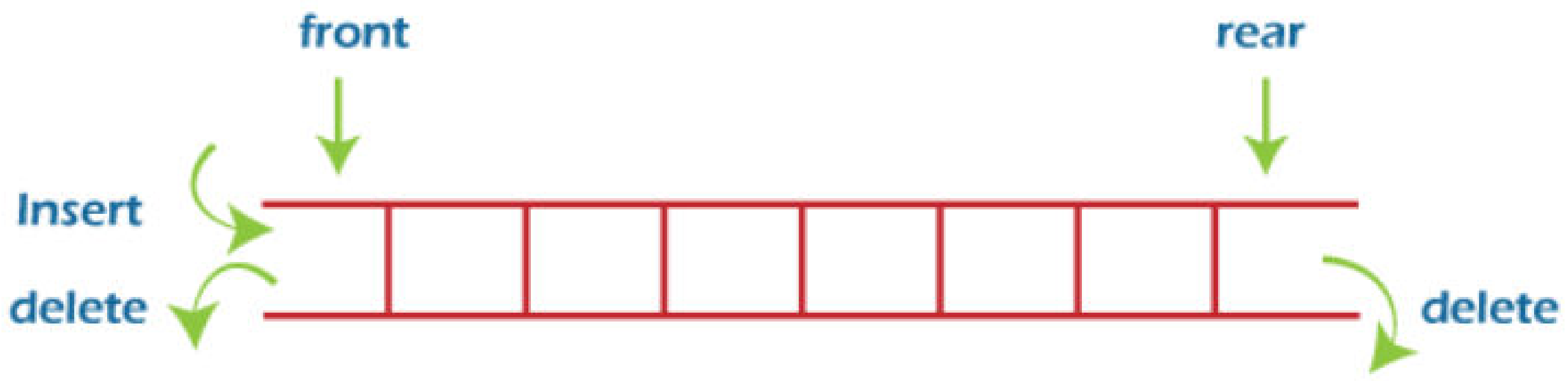
Types of deque

There are two types of deque -

1. Input restricted queue
2. Output restricted queue

Input restricted Queue

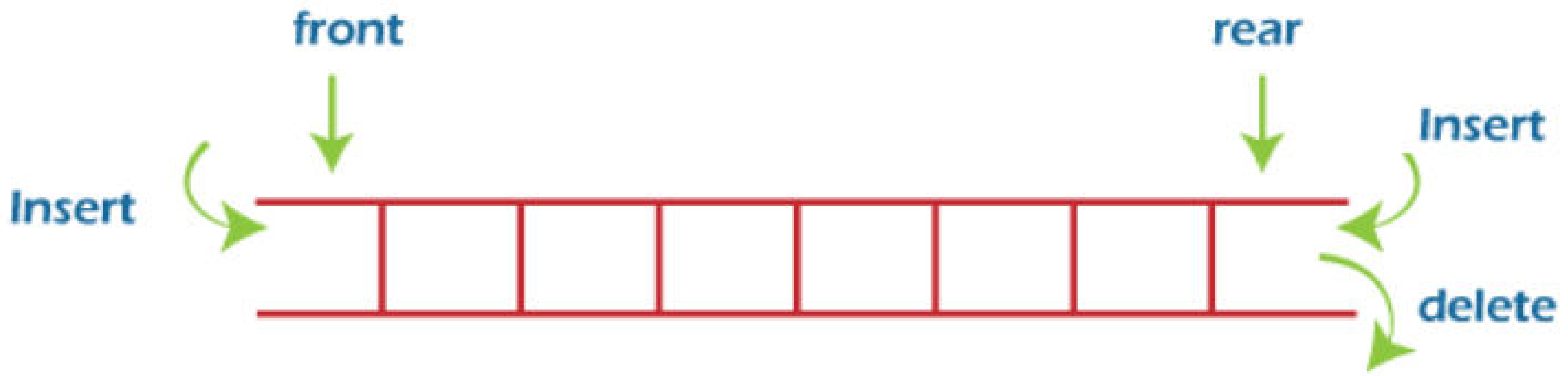
In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

Output restricted Queue

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue