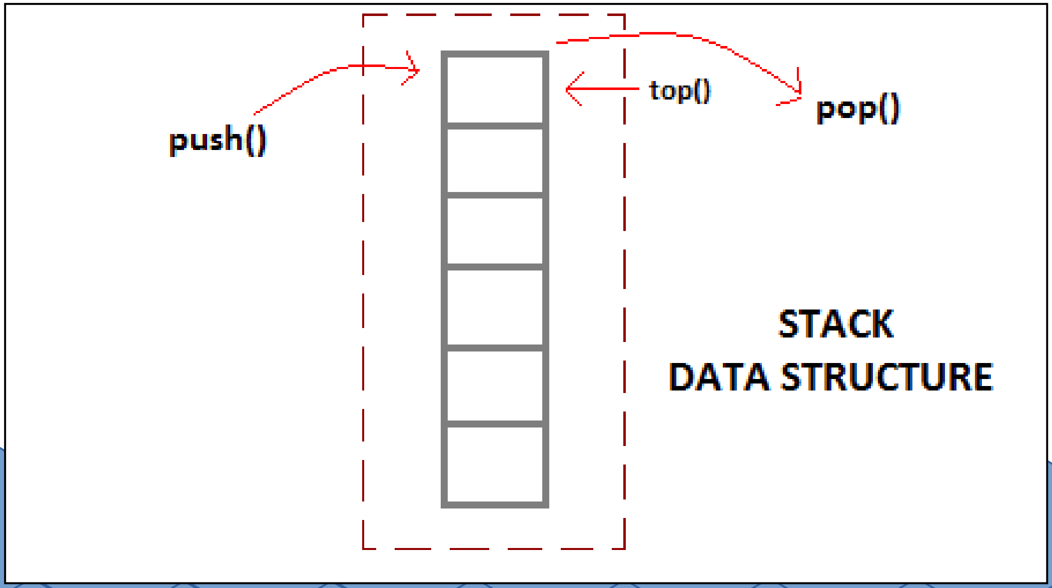
GLS UNIVERSITY

0301302 DATA STRUCTURES.
UNIT– II

STACK

- Stack is an abstract data type with a predefined capacity.
- A stack is a linear data structure.
- A stack is an ordered collection of data elements where the insertion and deletion operations take place at one end only
- Its called Last In First Out()
- It is a simple data structure that allows adding and removing elements in a particular order.
- Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.



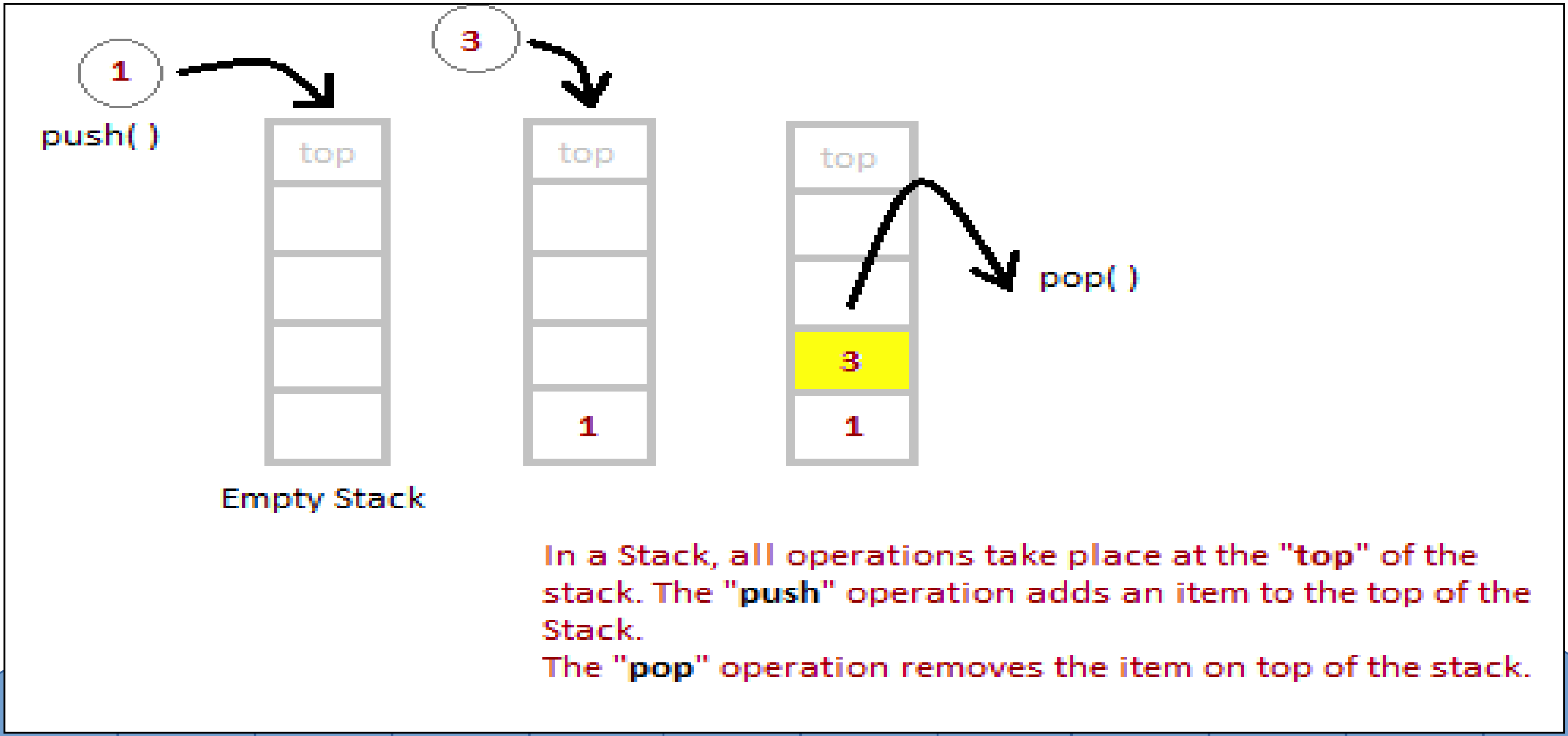
Stack

Basic features of Stack

- Stack is an ordered list of similar data type.
- Stack is a LIFO structure. (Last in First out) or FILO.
- push() function is used to insert new elements into the Stack and pop() is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called Top.
- Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

Implementation of Stack

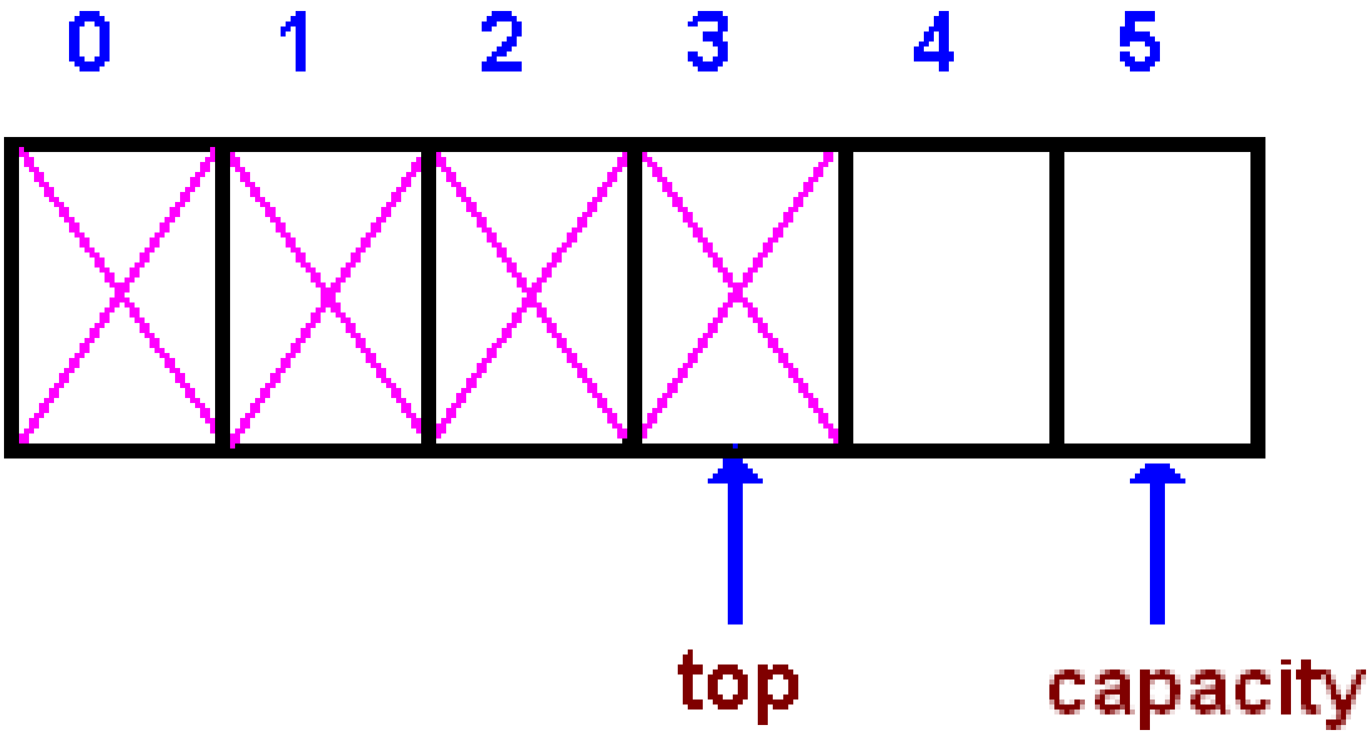
- Stack can be easily implemented using an Array or a Linked List.
- Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.



Array-based implementation

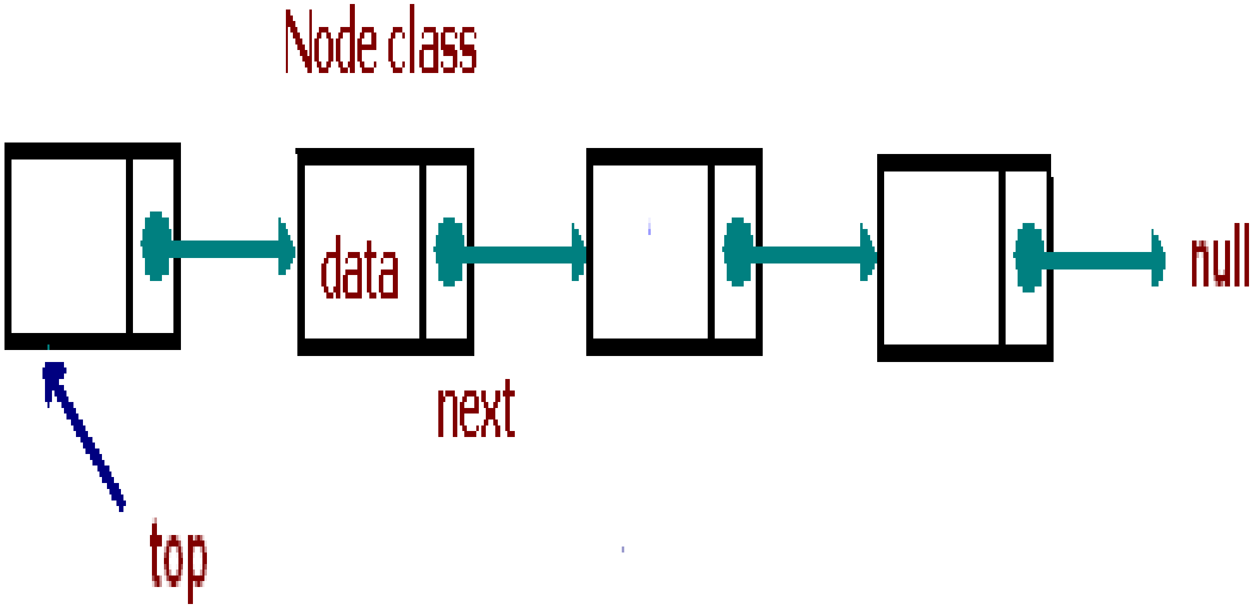
- In an array-based implementation we maintain the following fields: an array A of a default size (≥ 1), the variable top that refers to the top element in the stack and the capacity that refers to the array size.

.



Linked List-based implementation

- Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.



A Stack contains elements of same type arranged in sequential order.
All operations takes place at a single end that is top of the stack and following operations can be performed:

push(int data) – Insert an element at one end of the stack called top.

int pop() – Remove and return the element at the top of the stack, if it is not empty.

int top() – Return the element at the top of the stack without removing it, if the stack is not empty.

int size() – Return the number of elements in the stack.

boolean isEmpty() – Return true if the stack is empty, otherwise return false.

boolean isFull() – Return true if the stack is full, otherwise return false..

Basic Operations

- A stack is used for the following two primary operations −
  - **push()** − pushing (storing) an element on the stack.
  - **pop()** − removing (accessing) an element from the stack.

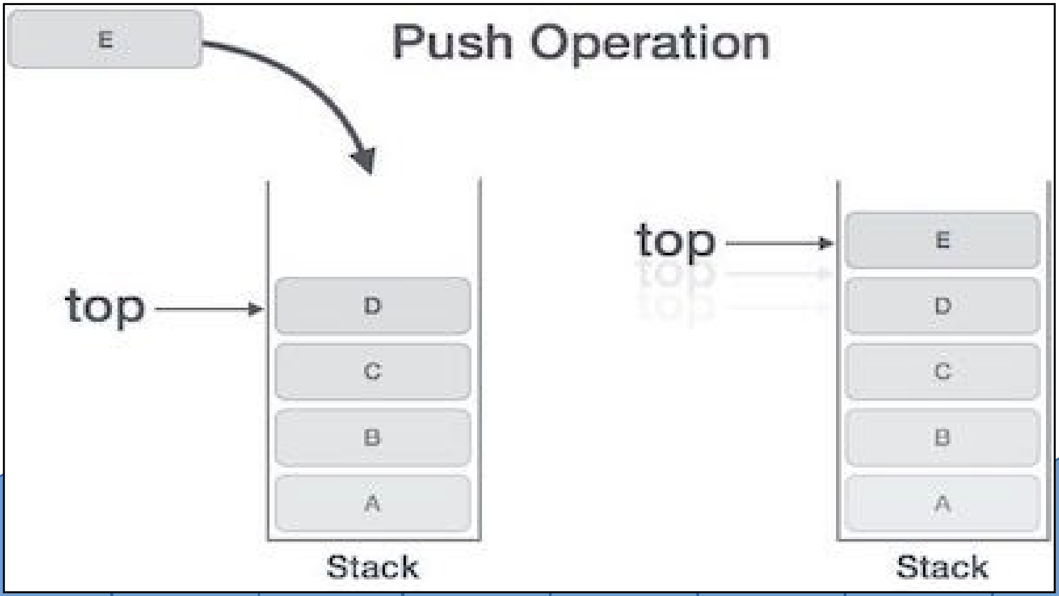
- To use a stack efficiently we need to check status of stack as well. For the same purpose, the following functionality is added to stacks −
  - **peep()** − get the top data element of the stack, without removing it.
  - **isFull()** − check if stack is full.
  - **isEmpty()** − check if stack is empty.

- At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it.

PUSH Operation

The process of putting a new data element onto stack is known as PUSH Operation. Push operation involves series of steps −

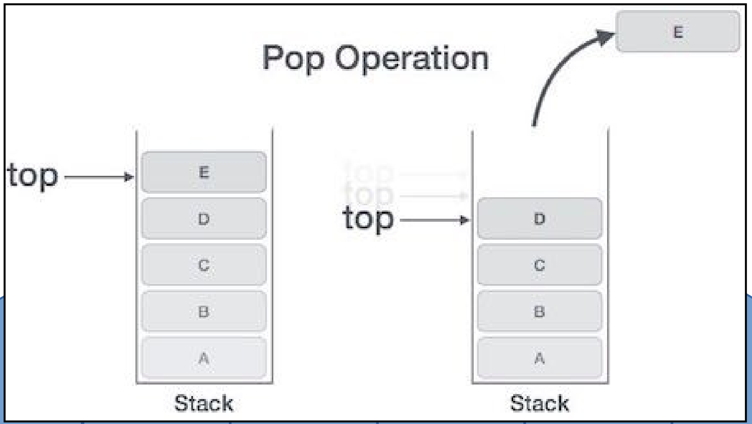
- Step 1 − Check if stack is full.
- Step 2 − If stack is full, produce error and exit.
- Step 3 − If stack is not full, increment top to point next empty space.
- Step 4 − Add data element to the stack location, where top is pointing.
- Step 5 − return success.



POP Operation

- Accessing the content while removing it from stack, is known as pop operation.
- In array implementation of pop() operation, data element is not actually removed, instead top is decremented to a lower position in stack to point to next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

- A POP operation may involve the following steps −
  - Step 1 − Check if stack is empty.
  - Step 2 − If stack is empty, produce error and exit.
  - Step 3 − If stack is not empty, access the data element at which top is pointing.
  - Step 4 − Decrease the value of top by 1.
  - Step 5 − return success.



Basic Operations: Push & Pop

Algorithm for PUSH Operation

Step 1: If TOP >= SIZE - 1 then

Write “Stack is Overflow”

Step 2: TOP = TOP + 1

Step 3: STACK [TOP] = X

Algorithm for POP Operation

Step 1: If TOP = -1 then

Write “Stack is Underflow”

Step 2: Return STACK [TOP]

Step 3: TOP = TOP - 1

Peek, isempty, isfull Operation

- **isfull():**
  - if top equals to MAXSIZE
  - return true
  - else
  - return false
  - endif
- **isempty():**
  - if top less than 1
  - return true
  - else
  - return false
  - endif
- **peep():**
  - return stack[top]

Applications

1)Stacks can be used for postfix expression evaluation.
2)Stacks can be used to check parenthesis matching in an expression.
3)Stacks can be used for Conversion from one form of expression to another.
4)Stacks can be used for Memory Management.
5)Stack data structures are used in backtracking problems.
(game playing, finding paths, exhaustive searching)
6) String Reversal
Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.
7)Processing function calls
8) simulating recursion

The way to write arithmetic expression is known as a notation.

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

These notations are −

1)Infix Notation

2)Prefix (Polish) Notation

3)Postfix (Reverse-Polish) Notation

INFIX

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

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

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.

The following table briefly tries to show the difference in all three notations –

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
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 -

1. Expression Evaluation

Stack data structure is used for evaluating the given expression. For example, consider the following expression

5 * (6 + 2) - 12 / 4

Since parenthesis has the highest precedence among the arithmetic operators, (6 +2) = 8 will be evaluated first. Now, the expression becomes

5 * 8 - 12 / 4

* and / have equal precedence and their associativity is from left-to-right. So, start evaluating the expression from left-to-right.

5 * 8 = 40 and 12 / 4 = 3

Now, the expression becomes 40 - 3

And the value returned after the subtraction operation is 37

Rules or Operator priorities

1. Exponentiation (^), Unary (+), Unary (–), and not (~
2. Multiplication (×) and division (/)
3. Addition (+) and subtraction (–)
4. Relational operators <, ≤ , =, ≠, ≥, >
5. Logical AND
6. Logical OR

Remember: If there is more than one ^ operator present in your expression, evaluate them from right to left.

For other same priority operators associativity is from left to right.

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)

As multiplication operation has precedence over addition, b * c will be evaluated first.

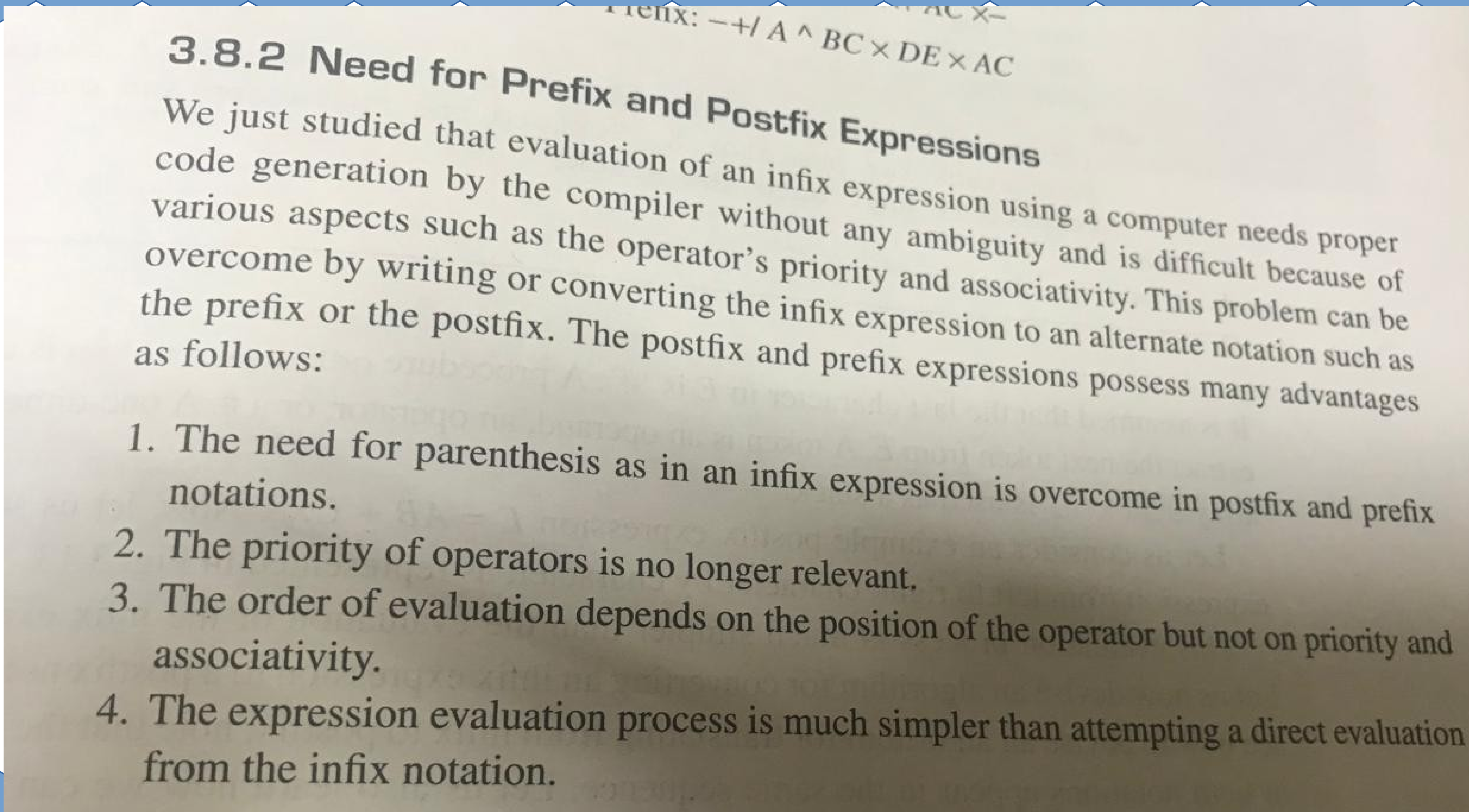
Associativity:

Associativity 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.

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

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (∗) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

To evaluate expression first it has to be converted into postfix notation



Prefix: $-+/A \wedge BC \times DE \times AC$

3.8.2 Need for Prefix and Postfix Expressions

We just studied that evaluation of an infix expression using a computer needs proper code generation by the compiler without any ambiguity and is difficult because of various aspects such as the operator's priority and associativity. This problem can be overcome by writing or converting the infix expression to an alternate notation such as the prefix or the postfix. The postfix and prefix expressions possess many advantages as follows:

1. The need for parenthesis as in an infix expression is overcome in postfix and prefix notations.
2. The priority of operators is no longer relevant.
3. The order of evaluation depends on the position of the operator but not on priority and associativity.
4. The expression evaluation process is much simpler than attempting a direct evaluation from the infix notation.

Infix expression 4 + 5 * 6

Consider the postfix expression 4 5 6 * +

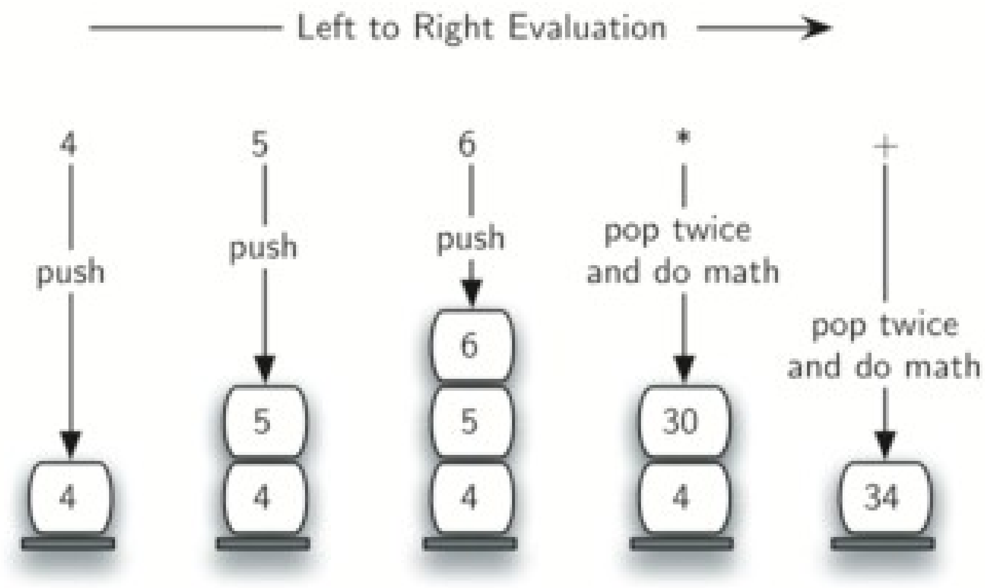
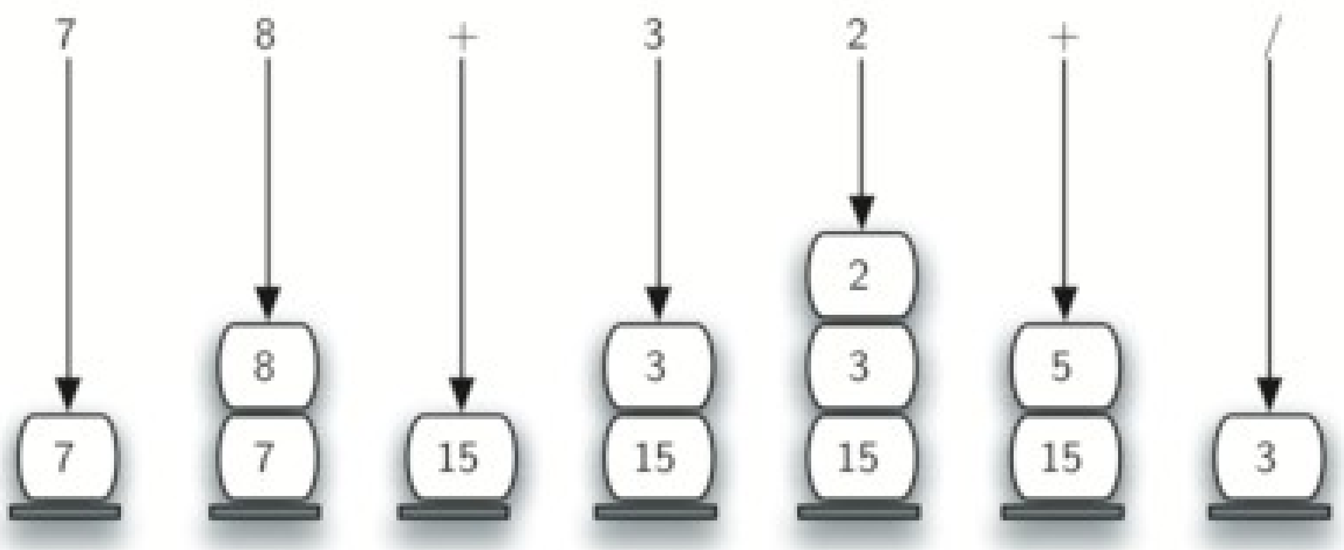


Figure 10: Stack Contents During Evaluation

7 8 + 3 2 + /



Postfix to infix

AB+C*

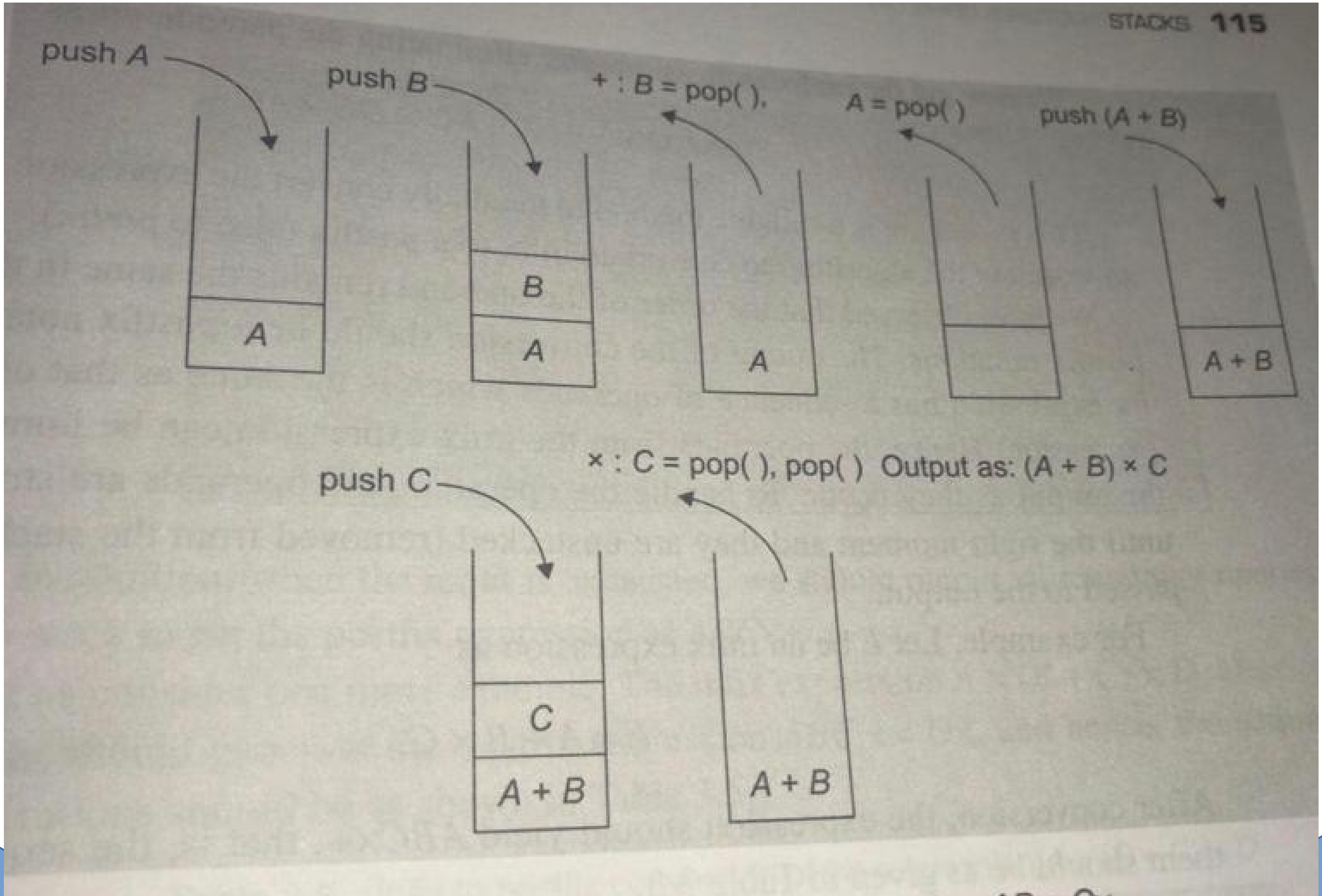


Fig. 3.13 Evaluation of postfix expression AB + C×

2. Parenthesis Matching

Given an expression, you have to find if the parenthesis is either correctly matched or not. For example, consider the expression (a + b) * (c + d).

In the above expression, the opening and closing of the parenthesis are given properly and hence it is said to be a correctly matched parenthesis expression. Whereas, the expression, (a + b * [c + d)

is not a valid expression as the parenthesis are incorrectly given.

3. Expression Conversion

Converting one form of expressions to another is one of the important applications of stacks.

1. Infix to Prefix
2. Infix to Postfix
3. Prefix to Infix
4. Prefix to Postfix
5. Postfix to Infix

Infix	Prefix	Postfix
a + b	+ b a	a b +
(a + b) * (c + d)	* + d c + b a	a b + c d + *
b * b - 4 * a * c	- * c * a 4 * b b	b b * 4 a * c * -

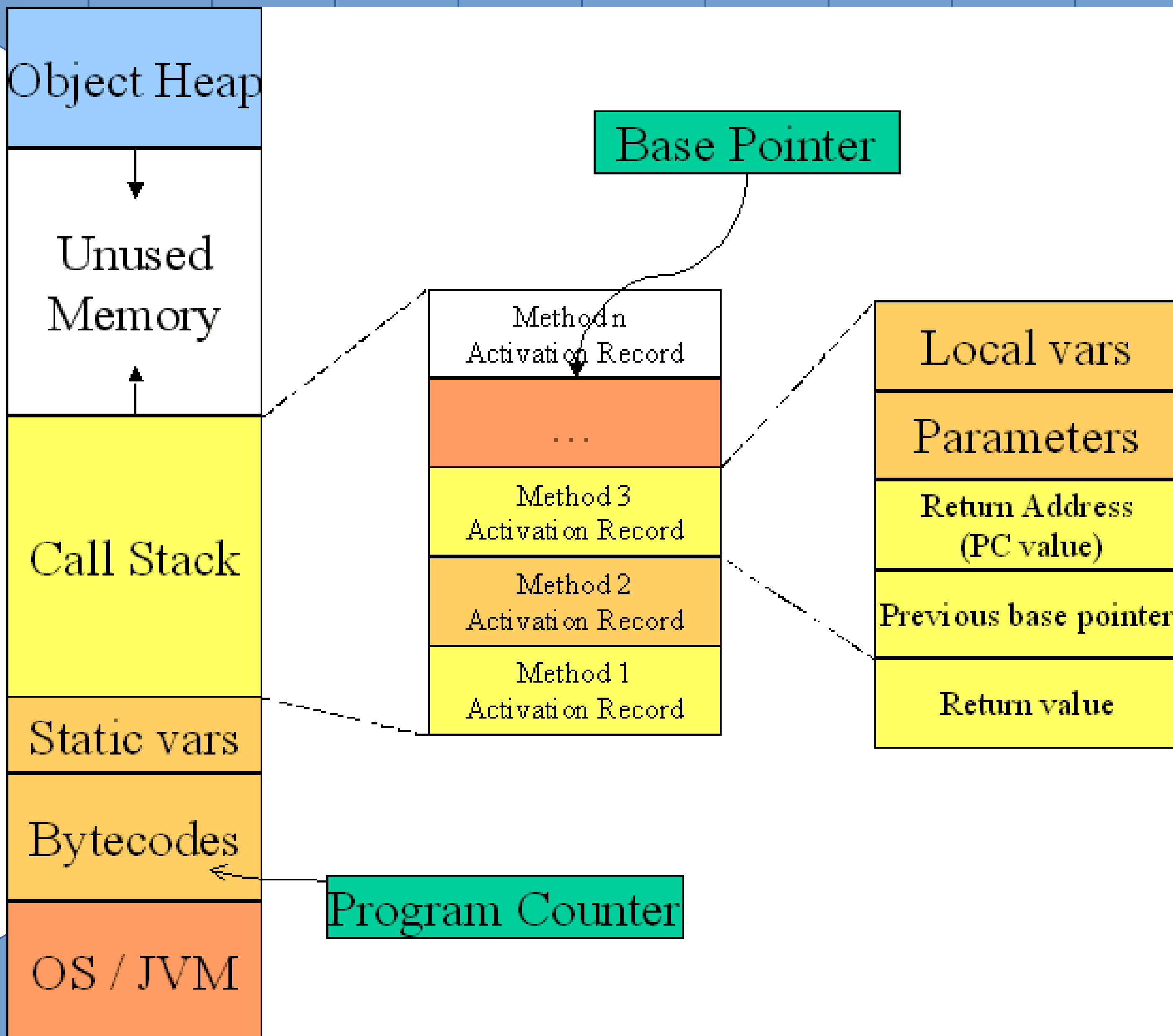
Infix Expression	Prefix Expression	Postfix Expression
A+B*C+D	++A*BCD	ABC*+D+
(A+B)*(C+D)	*+AB+CD	AB+CD+*
A*B+C*D	+*AB*CD	AB*CD*+
A+B+C+D	+++ABCD	AB+C+D+

Infix to postfix using stack.

Table 3.9 Infix to postfix conversion of the expression $A \wedge B \times C - C + D/A/(E + F)$

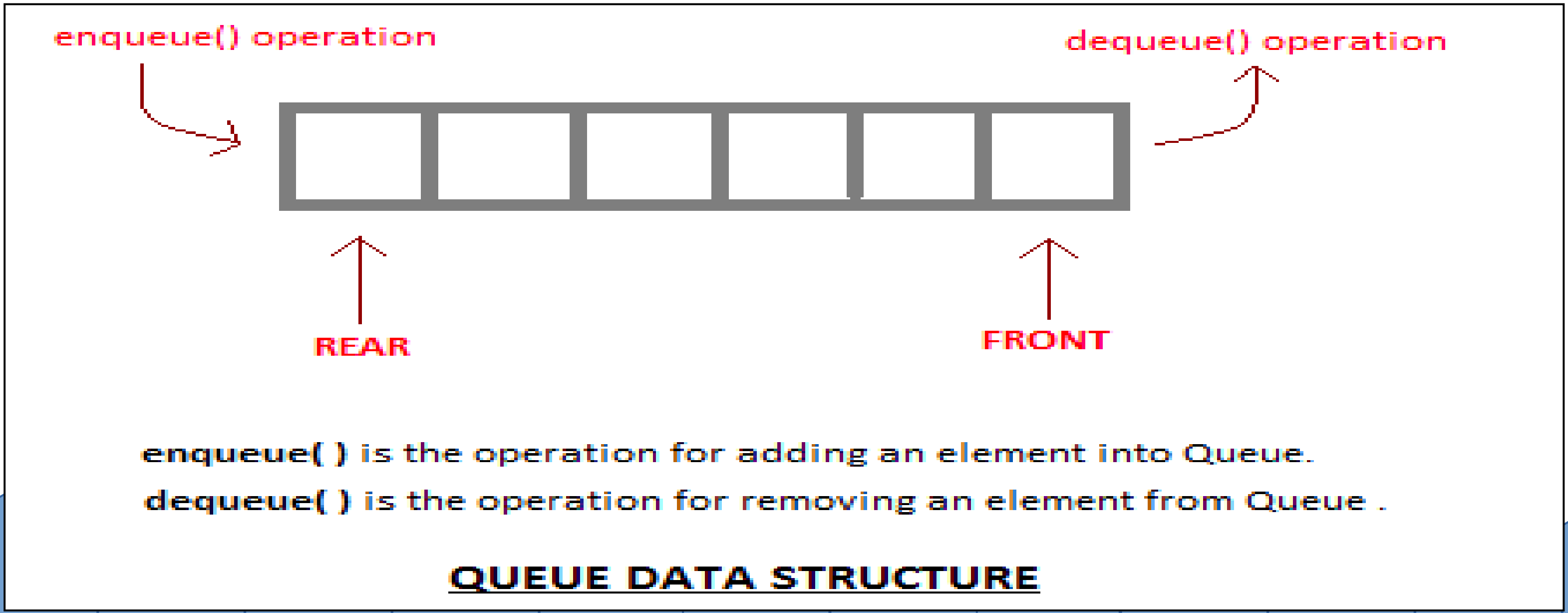
Character scanned	Stack contents	Postfix expression
A	Empty	A
$\wedge$	$\wedge$	A
B	$\wedge$	AB
$\times$	$\times$	$AB\wedge$
C	$\times$	$AB\wedge C$
$-$	$-$	$AB\wedge C\times$
C	$-$	$AB\wedge C\times C$
$+$	$+$	$AB\wedge C\times C-$
D	$+$	$AB\wedge C\times C-D$
$/$	$+/$	$AB\wedge C\times C-D$
A	$+/$	$AB\wedge C\times C-DA$
$/$	$+/$	$AB\wedge C\times C-DA/$
$($	$+/($	$AB\wedge C\times C-DA/$
E	$+/($	$AB\wedge C\times C-DA/E$
$+$	$+/(+$	$AB\wedge C\times C-DA/E$
F	$+/(+$	$AB\wedge C\times C-DA/EF$
$)$	$+/$	$AB\wedge C\times C-DA/EF+$
	Empty	$AB\wedge C\times C-DA/EF+/+$

4. Stacks can be used for Memory Management.



QUEUE

- Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called REAR(also called tail), and the deletion of exisiting element takes place from the other end called as FRONT(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.
- The process to add an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue.



Queue

Basic features of Queue

- Like Stack, Queue is also an ordered list of elements of similar data types.
- Queue is a FIFO(First in First Out) structure.
- Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
- peek( ) function is oftenly used to return the value of first element without dequeuing it.

Applications of Queue

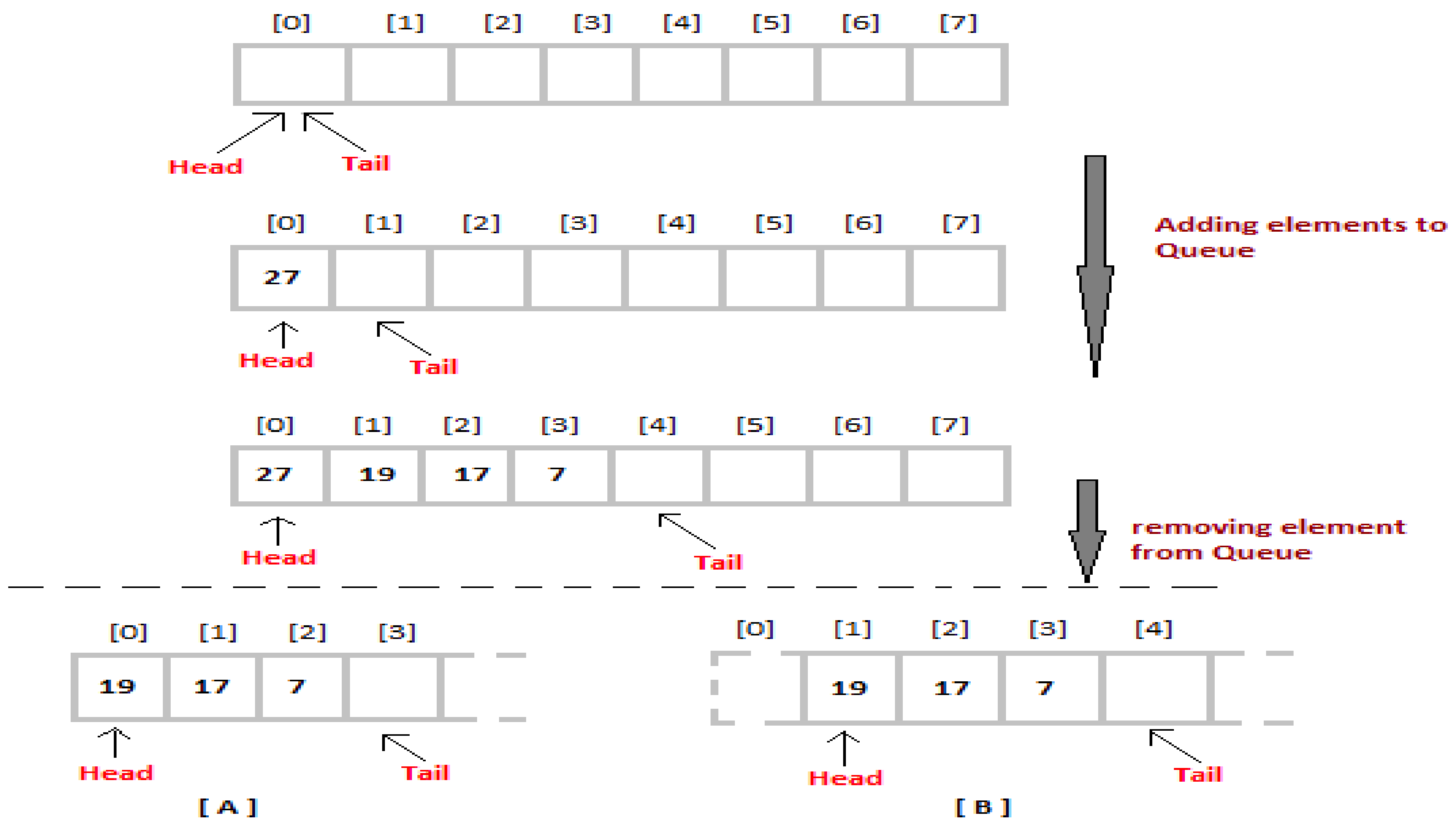
- Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :
  - Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
  - In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
  - Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.
  - A real world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world example can be seen as queues at ticket windows & bus-stops.

Implementation of Queue

- Queue can be implemented using an Array or Linked List. Initially the head(FRONT) and the tail(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.

- When we remove element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at head position, and then one by one move all the other elements on position forward. In approach [B] we remove the element from head position and then move head to the next position.

Implementation of Queue



Basic Operations

- Queue operations may involve initializing or defining the queue, utilizing it and then completing erasing it from memory. Here we shall try to understand basic operations associated with queues −
  - enqueue() − add (store) an item to the queue.
  - dequeue() − remove (access) an item from the queue.
- Few more functions are required to make above mentioned queue operation efficient. These are −
  - peek() − get the element at front of the queue without removing it.
  - isfull() − checks if queue is full.
  - isempty() − checks if queue is empty.
- In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in queue we take help of rear pointer.

Basic Operations

- **peek():**

```
return queue[front]
```

- **isfull():**

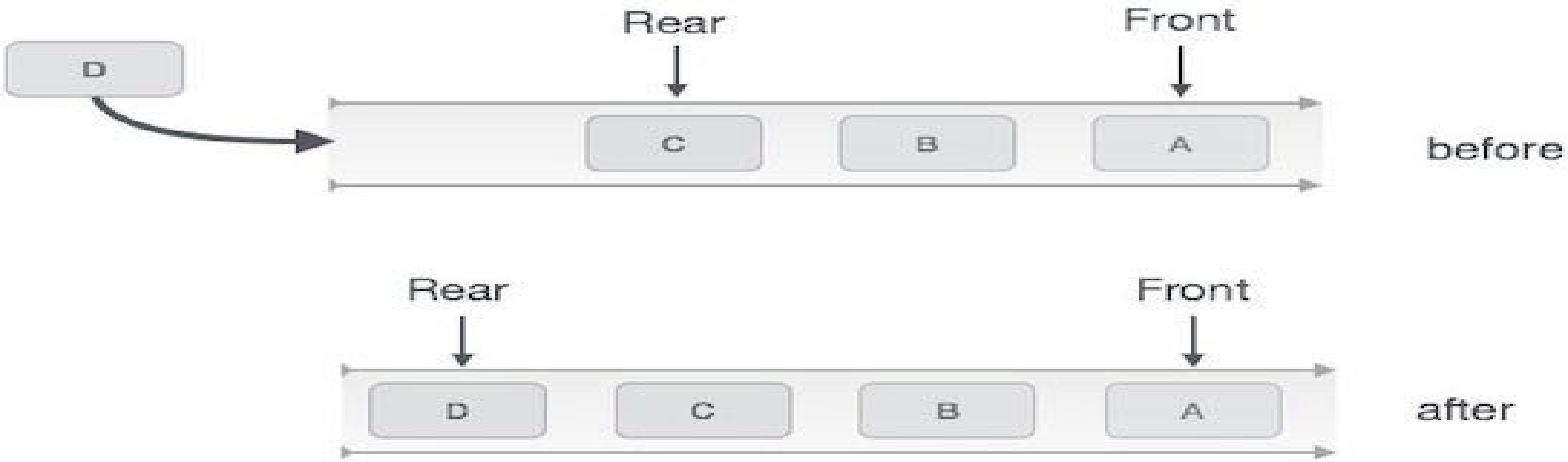
```
if(rear == MAXSIZE - 1)
    return true;
else
    return false;
```

- **isempty():**

```
if(front ==-1 && rear ==-1)
    return true;
else
    return false;
```

Enqueue Operation

- Step 1 − Check if queue is full.
- Step 2 − If queue is full, produce overflow error and exit.
- Step 3 − If queue is not full, increment rear pointer to point next empty space.
- Step 4 − Add data element to the queue location, where rear is pointing.
- Step 5 − return success.



Queue Enqueue

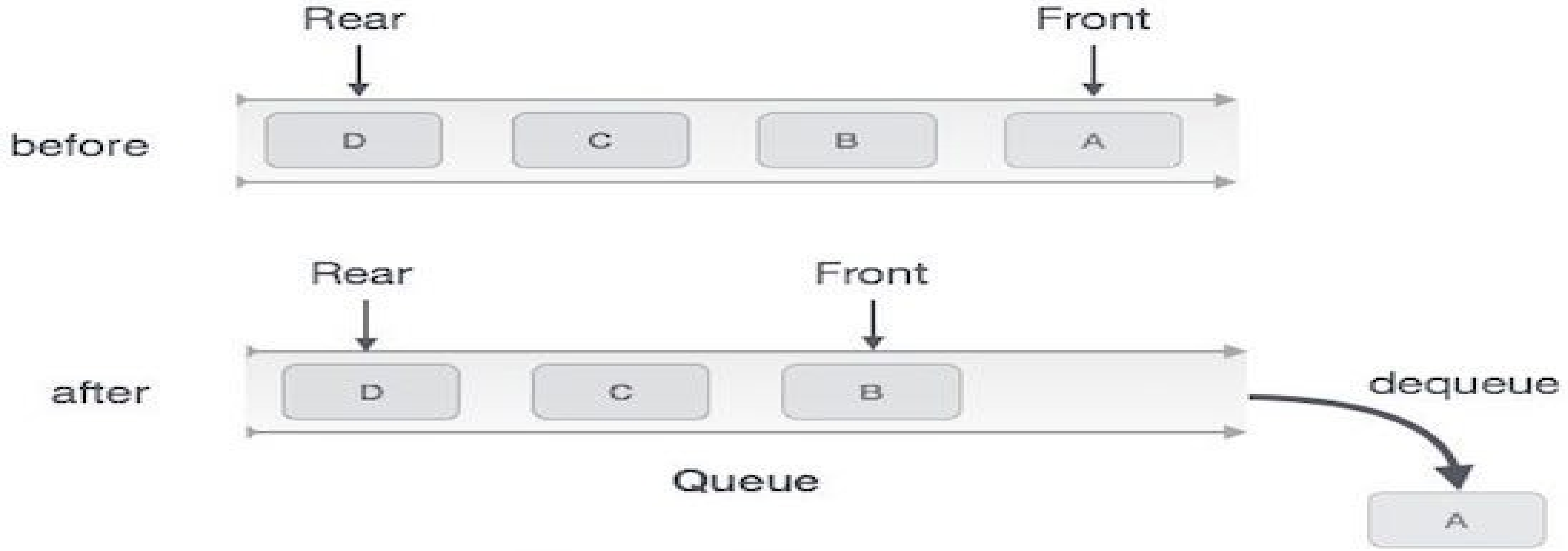
Enqueue Operation

Algorithm/Logic:

```
if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
```

Dequeue Operation

- Step 1 − Check if queue is empty.
- Step 2 − If queue is empty, produce underflow error and exit.
- Step 3 − If queue is not empty, access data where front is pointing.
- Step 3 − Increment front pointer to point next available data element.
- Step 5 − return success.



Queue Dequeue

Dequeue Operation

Algorithm/Logic:

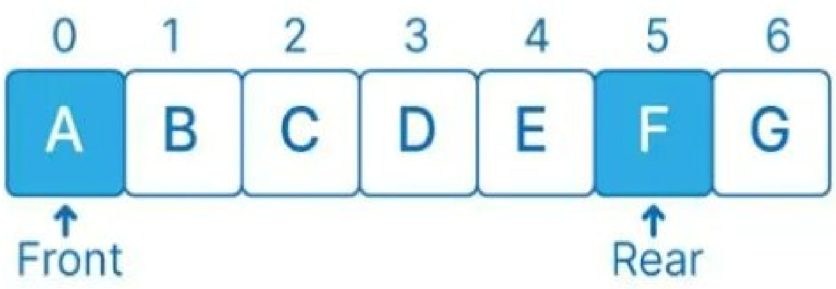
```
if(isempty())
    return 0;

  int data = queue[front];
  front = front + 1;

  return data;
```

Types of Queue

- **Queue can be of four types:**
  - **1. Simple Queue**
  - **2. Circular Queue**
  - **3. Priority Queue**
  - **4. Dequeue (Double Ended queue)**

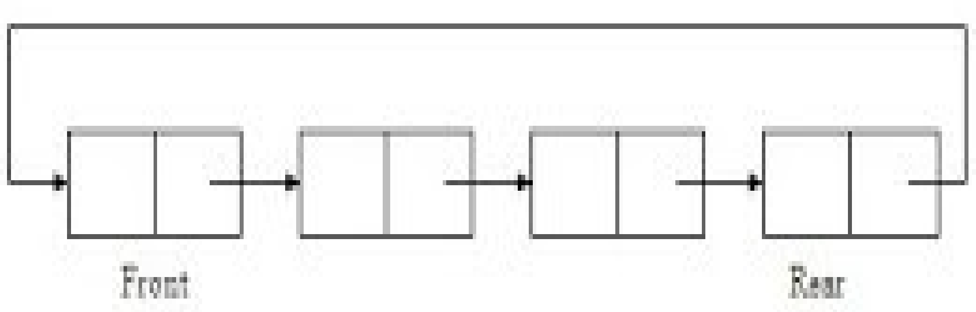
Types of Queue

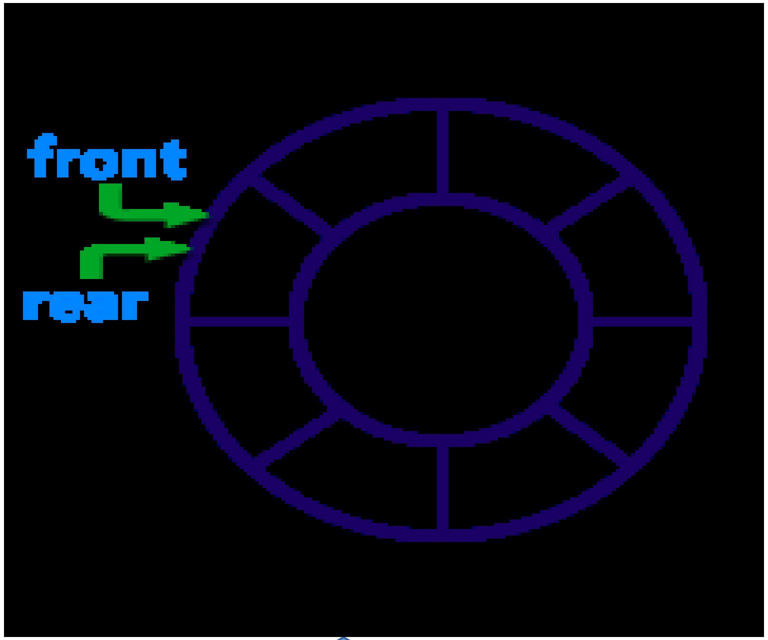
1. Simple Queue: In Simple queue Insertion occurs at the rear of the list, and deletion occurs at the front of the list.



Types of Queue

2. Circular Queue : A circular queue is a queue in which all nodes are treated as circular such that the first node follows the last node.
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.





Types of Queue

- In a normal Queue Data Structure, we can insert elements until queue becomes full. But once if queue becomes full, we can not insert the next element until all the elements are deleted from the queue. For example consider the queue below...
- After inserting all the elements into the queue.

Types of Queue

- Now consider the following situation after deleting three elements from the queue.

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.

Types of Queue

To implement a circular queue data structure using array, we first perform the following steps before we implement actual operations.

- **Step 1:** Include all the header files which are used in the program and define a constant 'SIZE' with specific value.
- **Step 2:** Declare all user defined functions used in circular queue implementation.
- **Step 3:** Create a one dimensional array with above defined SIZE (int cQueue[SIZE])
- **Step 4:** Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)
- **Step 5:** Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue

Types of Queue

- **enQueue(value) - Inserting value into the Circular Queue**
- In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...

- Step 1: Check whether queue is FULL. ((rear == SIZE-1 && front == 0) || (front == rear+1))
- Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
- Step 3: If it is NOT FULL, then check rear == SIZE - 1 && front != 0 if it is TRUE, then set rear = -1.
- Step 4: Increment rear value by one (rear++), set queue[rear] = value and check 'front == -1' if it is TRUE, then set front = 0.

Types of Queue

- **deQueue() - Deleting a value from the Circular Queue**
- In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position. The deQueue() function doesn't take any value as parameter. We can use the following steps to delete an element from the circular queue...

- Step 1: Check whether queue is EMPTY. (front == -1 && rear == -1)
- Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
- Step 3: If it is NOT EMPTY, then display queue[front] as deleted element and increment the front value by one (front ++). Then check whether front == SIZE, if it is TRUE, then set front = 0. Then check whether both front - 1 and rear are equal (front -1 == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

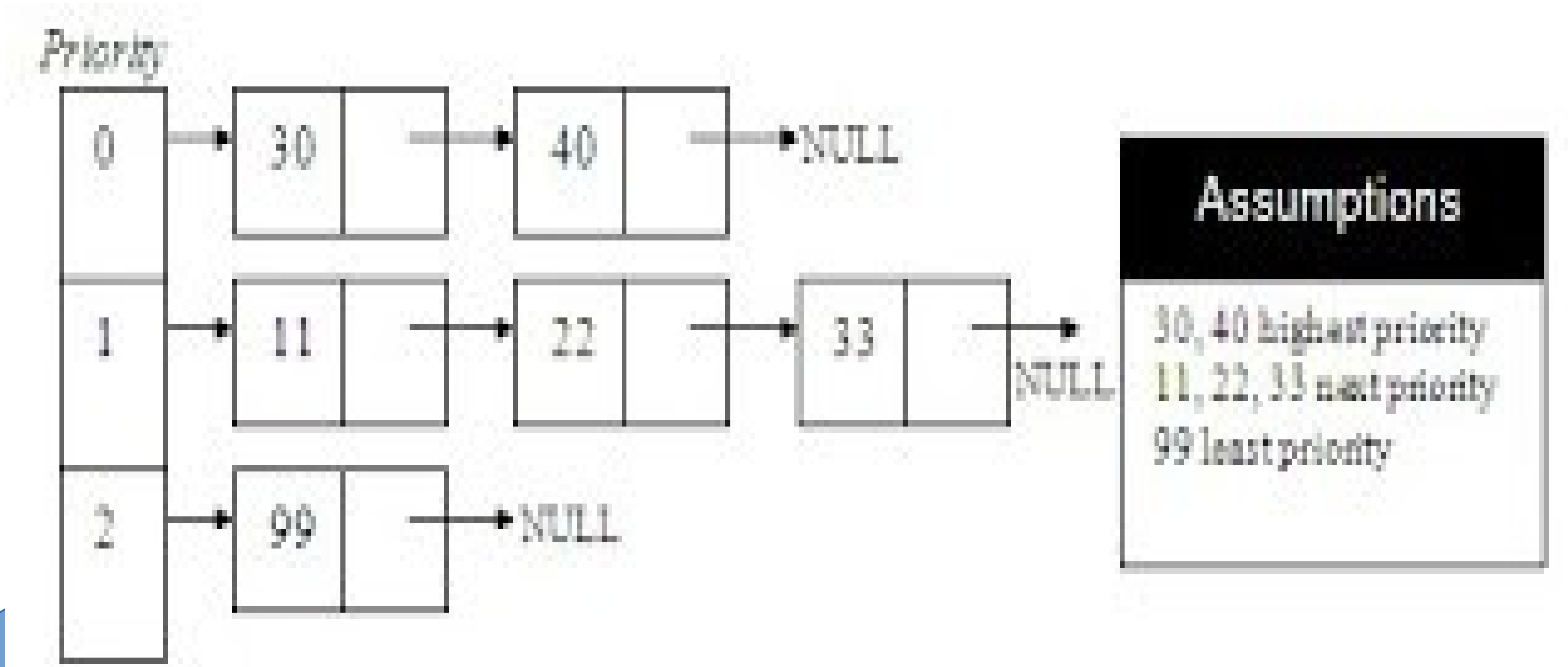
Types of Queue

- **display() - Displays the elements of a Circular Queue**
- We can use the following steps to display the elements of a circular queue...

- Step 1: Check whether queue is EMPTY. (front == -1)
- Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.
- Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front'.
- Step 4: Check whether 'front <= rear', if it is TRUE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.
- Step 5: If 'front <= rear' is FALSE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until'i <= SIZE - 1' becomes FALSE.
- Step 6: Set i to 0.
- Step 7: Again display 'cQueue[i]' value and increment i value by one (i++). Repeat the same until 'i <= rear' becomes FALSE

Types of Queue

3. Priority Queue: A priority queue is a queue that contains items that have some preset priority. When an element has to be removed from a priority queue, the item with the highest priority is removed first.



Types of Queue

4. Dequeue (Double Ended queue): In dequeue(double ended queue) Insertion and Deletion occur at both the ends i.e. front and rear of the queue.

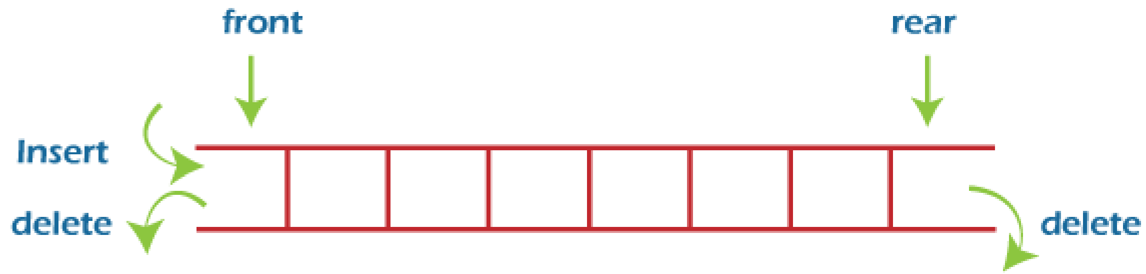
Two Types

Representation of deque

Input restricted queue
Output restricted queue

Input restricted Queue

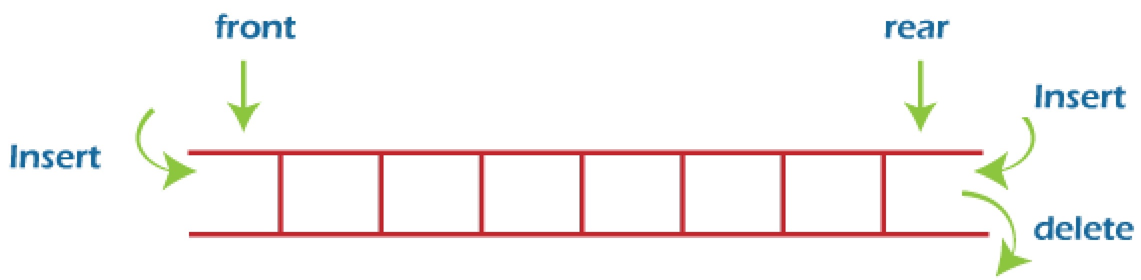
In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



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