Parul® University

Data Structure

Dr. Ghanshyam Rathod, Assistant Professor
IT and Computer Science

Parul®
University

DIGITAL LEARNING CONTENT

CHAPTER-2

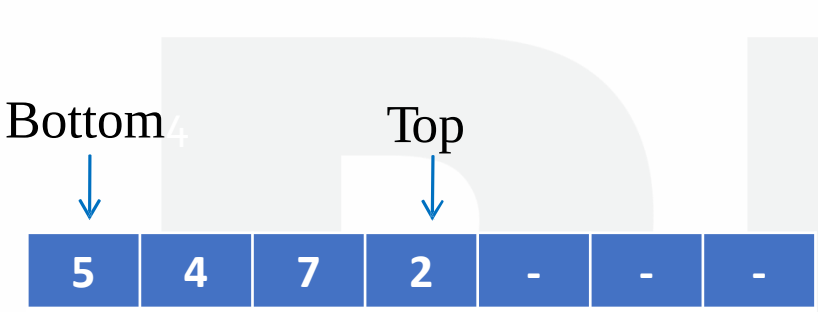
Stack and Queue

Stack

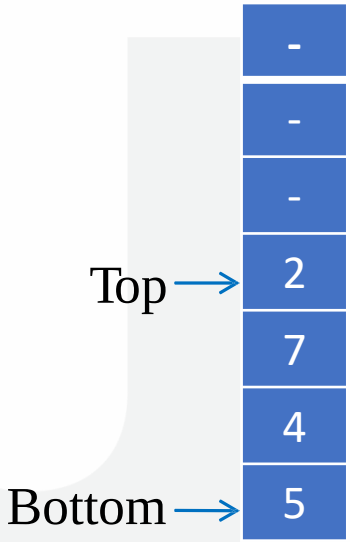
- Stacks are also an ordered collection of elements like array, but having a special feature that deletion and insertion of elements can be done only from one end called the top of the stack (TOP)

- Stack is sometimes called as Last-In-First-Out (LIFO) lists i.e. the element which is inserted first in the stack, will be deleted last from the stack.

Representation of stack in memory



Horizontal Representation



Vertical Representation

Stack- Examples

- Arrangements of disk plates in a rack
- Playing cards is example of stack
- Social media – Instagram posts

Terminology in stack

1. TOP:

- The pointer at the top most position of stack is called as TOP.
- It is the more commonly accessible element of stack.
- Insertion and deletion occur at top of the stacks.

2. BOTTOM

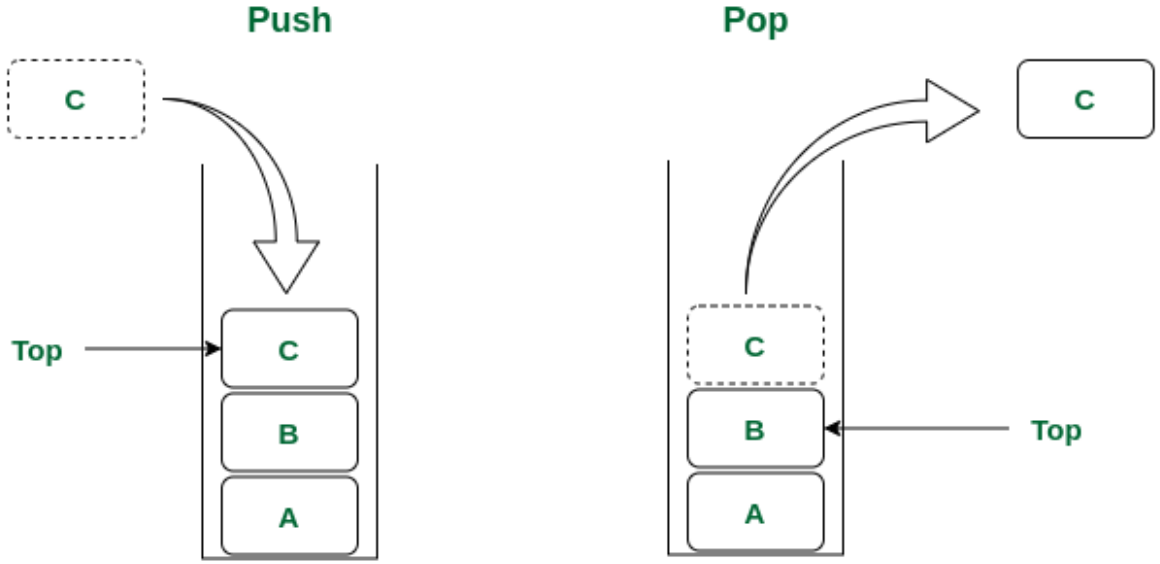
- The pointer at the last position of stack is called as BOTTOM.
- It is least commonly accessible element of the stack.

3. Stack overflow: If the stack is full and we are trying to insert new element into the stack, then system will give the stack overflow error.

4. Stack underflow: If the stack is empty and we are trying to delete element from the top of the stack then system will give the stack underflow error.

Stack Representation

- A stack can also be implemented by means of Array, Structure, Pointer, and Linked List.
- Stack can be a fixed size or dynamic.
- Now we are going to implement stack using arrays, which will make it a fixed size stack implementation.



Applications of Stack

- **Function calls:** Stacks are used to keep track of the return addresses of function calls, allowing the program to return to the correct location after a function has finished executing.
- **Recursion:** Stacks are used to store the local variables and return addresses of recursive function calls, allowing the program to keep track of the current state of the recursion.
- **Expression evaluation:** Stacks are used to evaluate expressions in postfix notation (Reverse Polish Notation).
- **Syntax parsing:** Stacks are used to check the validity of syntax in programming languages and other formal languages.
- **Memory management:** Stacks are used to allocate and manage memory in some operating systems and programming languages.

Pros of Stack

- **Simplicity:** Stacks are a simple and easy-to-understand data structure, making them suitable for a wide range of applications.
- **Efficiency:** Push and pop operations on a stack can be performed in constant time **(O(1))**, providing efficient access to data.
- **Last-in, First-out (LIFO):** Stacks follow the LIFO principle, ensuring that the last element added to the stack is the first one removed. This behavior is useful in many scenarios, such as function calls and expression evaluation.
- **Limited memory usage:** Stacks only need to store the elements that have been pushed onto them, making them memory-efficient compared to other data structures.

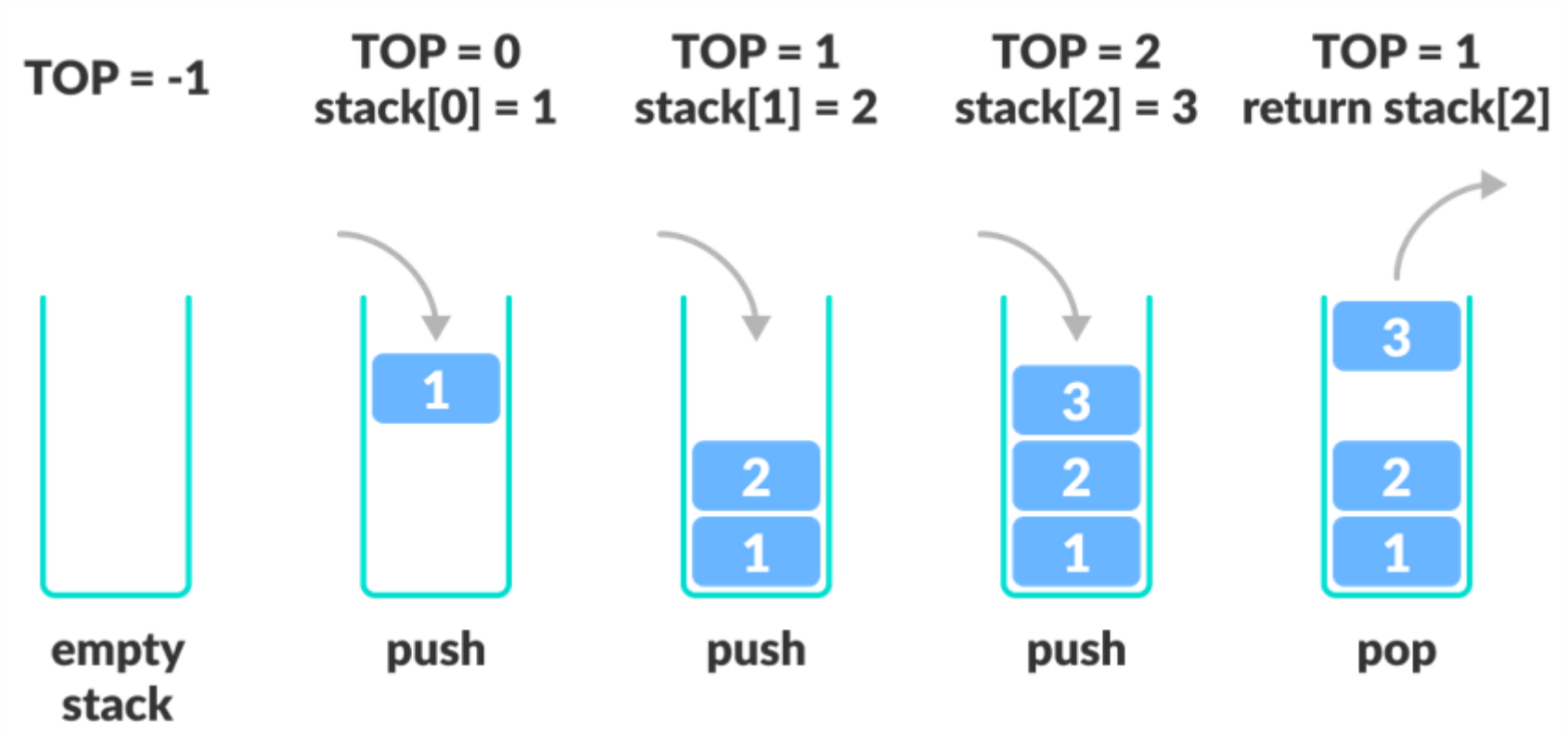
Cons of Stack

- **Limited access:** Elements in a stack can only be accessed from the top, making it difficult to retrieve or modify elements in the middle of the stack.
- **Potential for overflow:** If more elements are pushed onto a stack than it can hold, an overflow error will occur, resulting in a loss of data.
- **Not suitable for random access:** Stacks do not allow for random access to elements, making them unsuitable for applications where elements need to be accessed in a specific order.
- **Limited capacity:** Stacks have a fixed capacity, which can be a limitation if the number of elements that need to be stored is unknown or highly variable.

Operations on Stack

1. **PUSH :** To insert an item into the stack data structure
2. **POP :** To remove an item from a stack data structure.
3. **PEEP :** To find I th element from stack data structure.
4. **CHANGE :** To change I th element from stack data structure.

Representation of Stack with example



Algorithms – PUSH() - insert

- PUSH (Stack, TOP, Item)
- This procedure will insert an element Item into the stack which is represented by array containing N elements with the pointer TOP denoting top element in the stack.

```
Step 1: [Check for stack overflow?]
        If TOP >= N then
        write('stack overflow')
       Exit
 Step 2: [Increment the TOP pointer]
       TOP <-TOP +1
Step 3: [Insert an element into the stack]
      Stack[TOP] <-Item
Step 4: [Finished]
        Return
```

Algorithms – POP()

- POP(Stack, TOP, Item)
- This algorithm deletes an element Item from the top of a stack S containing N elements.

```
Step 1: [Check for stack underflow?]
        if TOP == 0 then
                write ('Stack Underflow')
        Exit
Step 2: [Accessing the value to be deleted]
        Item <- Stack[TOP]
Step 3: [Decrement the stack pointer]
        TOP<-TOP-1
Step 4: [Return deleted element Item ]
        Return Item
```

Algorithms – PEEP()

- PEEP(Stack, TOP, i, item)
- Given an array Stack containing N elements. Pointer TOP elements top element of the stack. This algorithm fetched the ith element in the value item.

  Step 1: [Check for stack underflow?]

  If (TOP-i+1) <= 0 then

  write('Stack underflow')

  Exit

  Step 2: [Storing the i^{th} element in item]

  item <- Stack[TOP-i+1]

  Step 3: [Finished]

  Return item

e.g. of peep operation

5
4
3
2
1

If we want to obtain 3rd element from the top of the stack.

Top=5 1=3

Result = top-1+1

= 5-3+1

= 3 <- address

S [3] returns 3

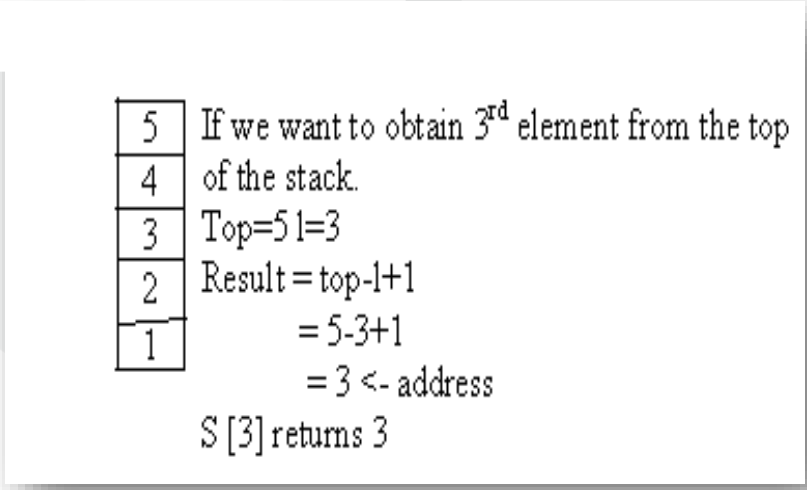
Algorithms – CHANGE()

- CHANGE (S, TOP, i, X):
- This algorithm updates the ith element with the value item in a stack Stack containing N elements.

  Step 1: [Check for stack underflow?]
  if (TOP-i+1 <= 0) then
  write('Stack underflow on change')
  Exit

  Step 2: [Changing the i^{th} element in stack]
  Stack[TOP-i+1] <- item

  Step 3: [Finished]
  Return

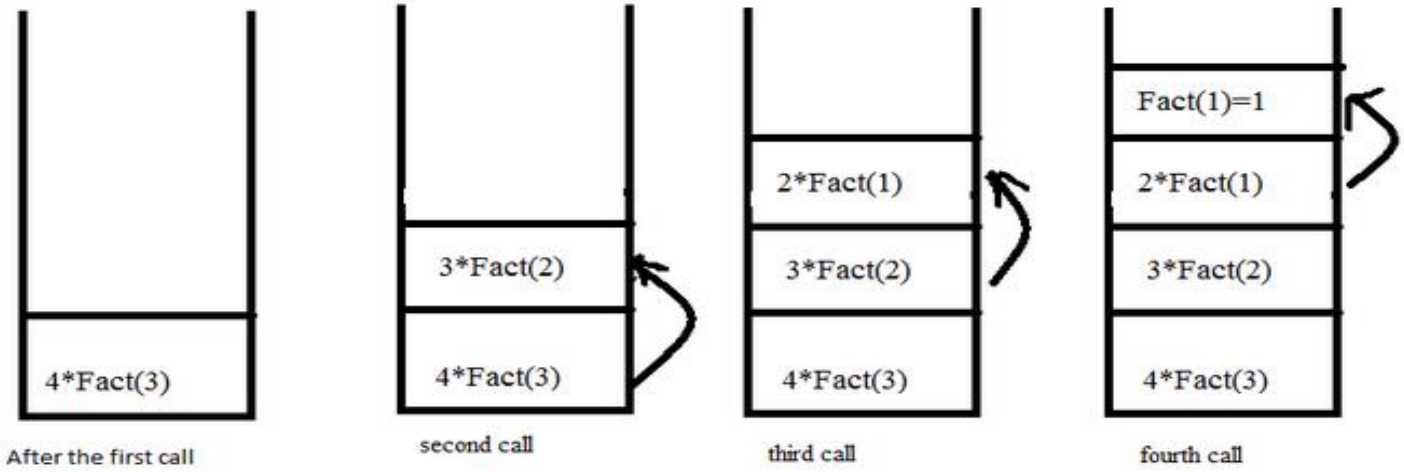


Recursion

- Recursion is the technique of defining a set or process in term of itself.
- E.g. The factorial function can be recursively defined as follows:

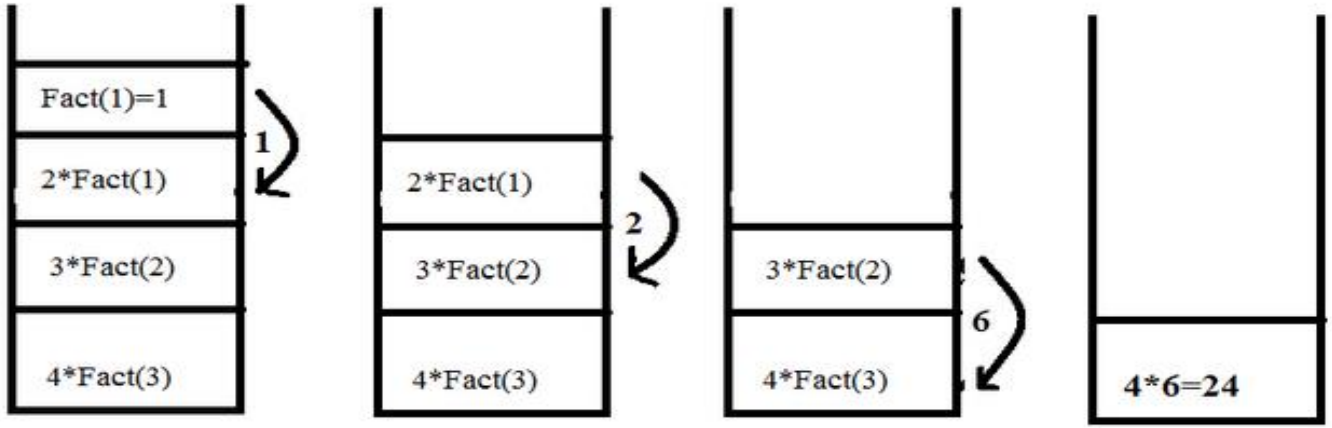
```
int factorial(int n)
{
    int fact;
    if(n==1)
        return(1);
    else
        fact = n*factorial(n-1);
        return(fact);
}
```

When function call happens previous variables gets stored in stack



After the first call | second call | third call | fourth call

Returning values from base case to caller function



Polish expressions and their compilations

1. Infix notation:

- The arithmetic operators appears between the operands.
- E.g. A+B*C

2. Suffix (postfix) notation or Reverse Polish Notation:

- The arithmetic operators appears after the operands
- E.g. A+B*C can be represented as ABC*+

3. Prefix notation or Polish notation:

- The arithmetic operators appears before the operands
- E.g. A+B*C can be represented as +*ABC

Precedence Table

Symbol / Operator	Precedence	Description
(), []	1	Parentheses, Array Subscript
↑	2	Exponent, Ex: 9^4 Base = 9, Exponent = 4
*, /, %	3	Multiplication, Division, Modulus
+, –	4	Addition, Subtraction

Infix Expression:

a + b / c – d * e

Postfix Expression:

a + b / c – d * e
a + bc/ – d * e
a + bc/ – de*
abc/+ – de*
abc/+de*–

Prefix Expression:

a + b / c – d * e
a + /bc – d * e
a + bc/ – *de
+abc/ – *de
– +abc/*de

Infix Expression:

(a + b) * (c – d)

Postfix Expression:

(a + b) * (c – d)

ab+ * (c – d)

ab+ * cd-

ab+cd-*

Prefix Expression:

(a + b) * (c – d)

+ab * (c – d)

+ab * –cd

*+ab–cd

Infix Expression:

(a + b ↑ c ↑ d) * (e + f / d)

Postfix Expression:

(a + b ↑ c ↑ d) * (e + f / d)
(a + bc ↑ ↑d) * (e + f / d)
(a + bc ↑d ↑) * (e + f / d)
abc ↑d ↑+ * (e + f / d)
abc ↑d ↑+ * (e + fd/)
abc ↑d ↑+ * efd/+
abc ↑d ↑+ efd/+*

Prefix Expression:

(a + b ↑ c ↑ d) * (e + f / d)
(a + ↑bc ↑ d) * (e + f / d)
(a + ↑ ↑bcd) * (e + f / d)
+a ↑ ↑bcd * (e + f / d)
+a ↑ ↑bcd * (e + /fd)
+a ↑ ↑bcd * +e/fd
*+a ↑ ↑bcd+e/fd

Precedence Table (For Algorithm)

Symbol / Operator	Input Precedence Function ***f***	Stack Precedence Function ***g***	Rank Function ***r***
+, –	1	2	-1
*, /, %	3	4	-1
↑	6	5	-1
Variables	7	8	1
(	9	0	-
)	0	-	-

Conversion of infix to postfix (Parenthesized)

Algorithm : REVPOL. Given an input string INFIX containing an infix expression which has been padded on the right with ‘)’ and whose symbols have precedence values given by Table, a vector S, used as a stack, and a function NEXTCHAR which, when invoked, returns the next character of its argument, this algorithm converts INFIX into reverse Polish and places the result in the string POLISH. The integer variable TOP denotes the top of the stack. Algorithms PUSH and POP are used for stack manipulation. The integer variable RANK accumulates the rank of the expression. Finally , the string variable TEMP is used for temporary storage purposes.

1.[Initialize stack]

TOP <- 1

S[TOP] <- ‘(‘

2. [Initialize output string and rank count]

POLISH <- “

RANK <- 0

Conversion of infix to postfix (Parenthesized)

3. [Get first input symbol]
   NEXT <- NEXTCHAR(INFIX)
4. [Translate the infix expression]
   Repeat thru step 7 while NEXT ⇦ “)”
5. [Remove symbols with greater precedence from stack]
   If TOP < 1
   then Write(‘INVALID’)
   Exit
   Repeat while f(NEXT) < g(S[TOP])
   TEMP <- POP(S,TOP)
   POLISH <- POLISH O TEMP
   RANK <- RANK + r(TEMP)
   If RANK < 1
   then Write(‘INVALID’)
   Exit

Conversion of infix to postfix (Parenthesized)

6. [Are there matching parentheses?]
   If f(NEXT) ⇄ g(S[TOP])
   then Call PUSH(S,TOP,NEXT)
   else POP(S,TOP)
7. [Get next input symbol]
   NEXT <- NEXTCHAR(INFIX)
8. [Is the expression valid?]
   If TOP ⇄ 0 or RANK ⇄ 1
   then Write('INVALID')
   else Write('VALID')
   Exit

Infix Expression:

(a + b ↑ c ↑ d) * (e + f / d))

Character Scanned	Contents of Stack	Reverse-Polish Expression	Rank
	(		
(	((		
a	((a		
+	((+	a	1
b	((+b	a	1
↑	((+↑	ab	2
c	((+↑c	abc	2
↑	((+↑↑	abc	3
d	((+↑d	abc↑d↑+	3
)	(	abc↑d↑+	1
*	(*	abc↑d↑+	1
(	(*(	abc↑d↑+	1
e	(*(e	abc↑d↑+e	1
+	(*(+	abc↑d↑+e	2
2	(*(+f	abc↑d↑+ef	2
/	(*(+/	abc↑d↑+ef	3
d	(*(+/d	abc↑d↑+efd/+	3
)	(*	abc↑d↑+efd/+*	2
)			1

Queue

A queue is a linear list of elements in which deletions can take place only at one end, called the front and insertions can take place only at other end, called the rear.

Queues are also called first-in-first-out (FIFO) lists OR First-Come-First-Serve (FCFS).

Terminologies and Real Examples of Queue

Terminologies:

1. REAR : Pointer representing the tail end of queue
2. FRONT : Pointer representing the front end of queue

Real Examples:

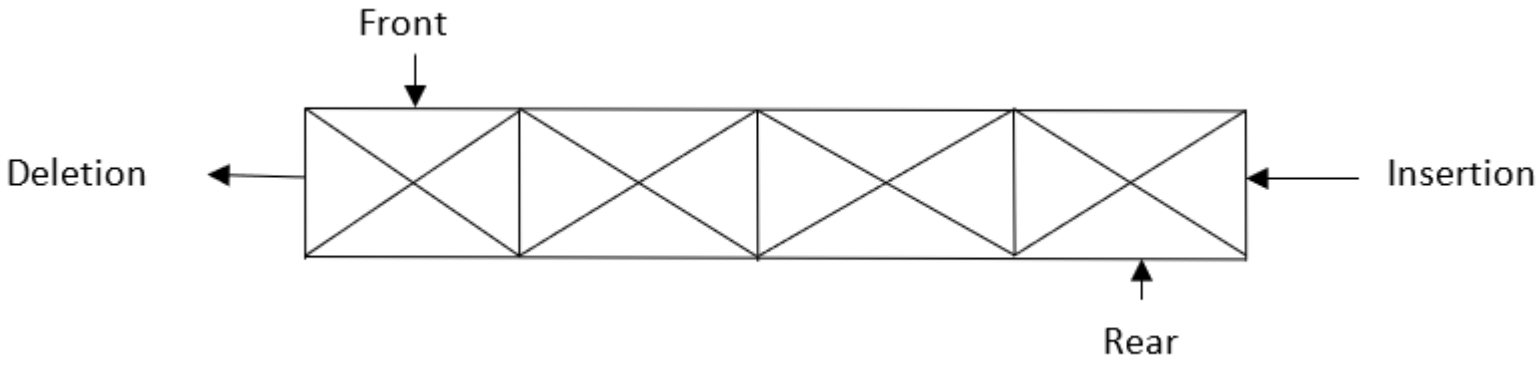
- Line of people at ticket window
- Line of vehicles at railway crossing

Types of Queue:

1. Simple Queue
2. Circular Queue
3. Double Ended Queue
4. Priority Queue

Simple / Linear Queue:

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



Algorithm to INSERT an element in a simple queue

QINSERT(Q,F,N,R,Y)

Step 1 : [Check for an overflow]
    if R>=N then
        write('Overflow')
        return
Step 2 : [increment rear pointer]
        R ← R + 1
Step 3 : [insert an element at rear position]
    Q[R] ← Y
Step 4 : [is front pointer properly set ?]
    if F=0 then
        F ← 1
    return

Algorithm to DELETE an element in a simple queue

QDELETE(Q,F,R,Y)

Step 1 : [Check for an underflow]

if F = 0 then

write('Underflow')

return

Step 2 : [delete an element]

Y ← Q[F]

Step 3 : [is the queue empty ?]

if F=R then

F ← R ← 0

else

F ← F+1

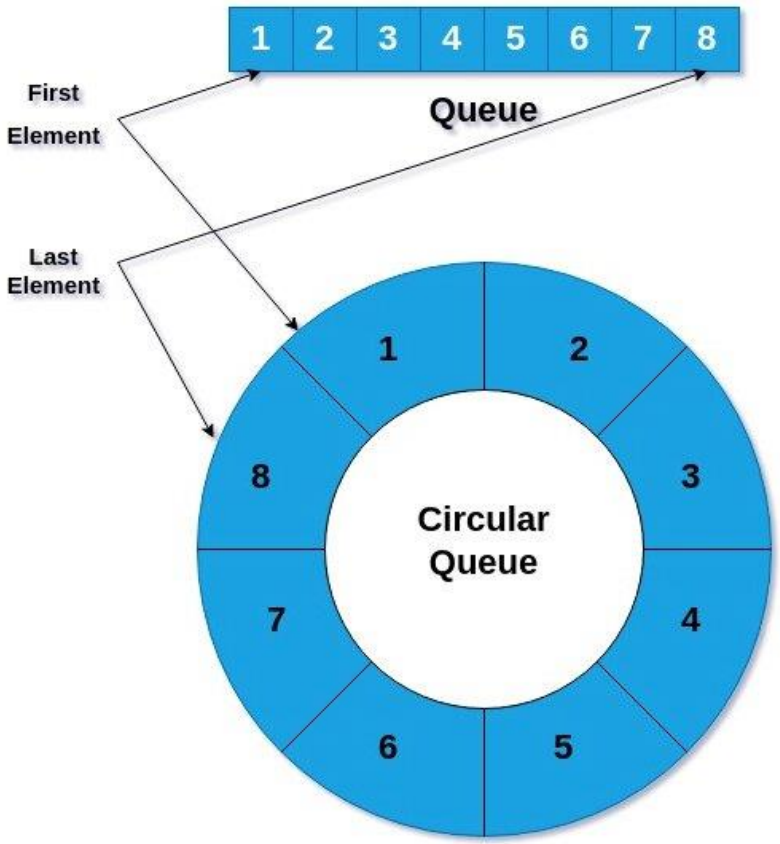
Step 4 : [Return deleted element]

return (Y)

Circular Queue:

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

- The queue in which elements are arranged in a circular fashion
- In a circular queue any element is accessible from any position but only in a forward manner.



Algorithm to INSERT an element in a circular queue

QINSERT(F,R,N,Q,Y)

Step 1: [Reset rear pointer ?]

if R = N

then R ← 1

Else

R ← R + 1

Step 2 : [Check for overflow]

if R = F

then write('OVERFLOW')

Return

Step 3 : [Insert element]

Q[R] ← Y

Step 4 : [Is front pointer properly set?]

if F=0

then F ← 1

Return

Algorithm to DELETE an element in a circular queue

QDELETE (F,R,Q,Y)

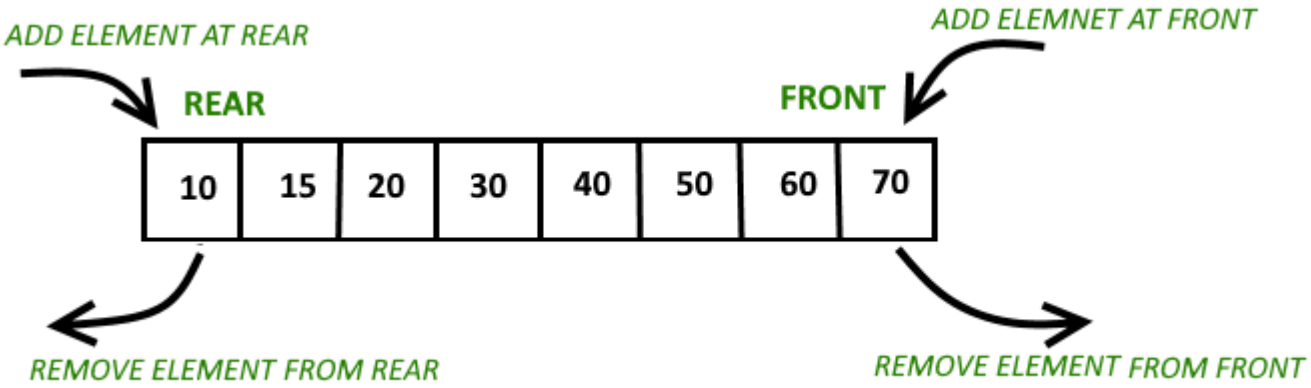
Step 1 : [Underflow ?]
    if F = 0
        then write('Underflow … No elements to delete')
    Return
Step 2 : [Delete an element]
    Y ← Q[F]
Step 3 : [Queue empty]
    if R = F
        then R ← F ← 0
    return (Y)
Step 4 : [Increment front pointer]
    if F = N
        then f ← 1
    else
        F ← F + 1
    Return (Y)

Double Ended Queue:

A D-queue is a linear list in which elements can be inserted or deleted from either end of a queue.

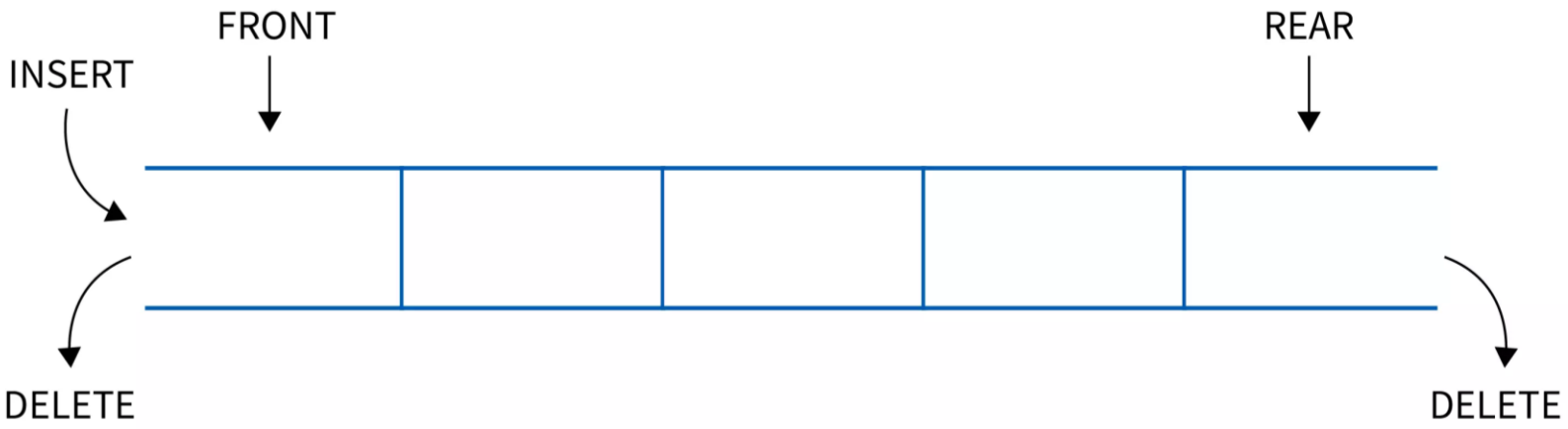
There are two variations in D-queue:

1. An input restricted D-queue
2. An output restricted D-queue



Input Restricted Double Ended Queue:

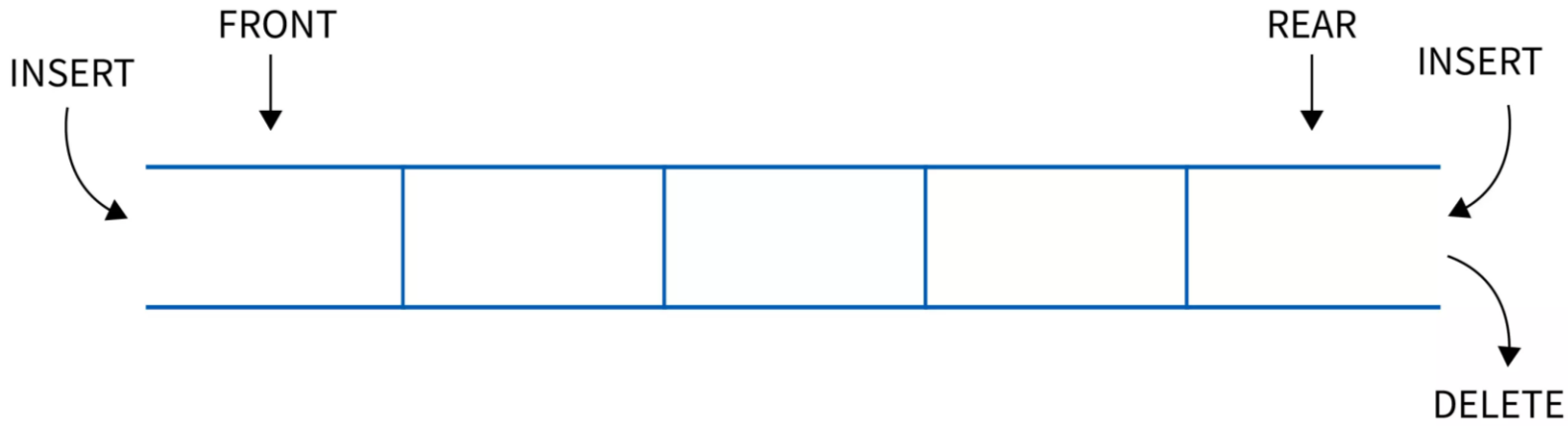
It allows insertion only at one end of the queue but deletion at both the ends of a queue.



INPUT RESTRICTED DOUBLE ENDED QUEUE

Output Restricted Double Ended Queue:

It allows deletion only at one end of the queue but insertion at both the ends of a queue.



OUTPUT RESTRICTED DOUBLE ENDED QUEUE

Double Ended queue (D-queue)

Here are four basic operations in usage of Deque that we will explore:

1. Insertion at rear end
2. Insertion at front end
3. Deletion at front end
4. Deletion at rear end

Algorithm for Insertion at rear end for D-Queue

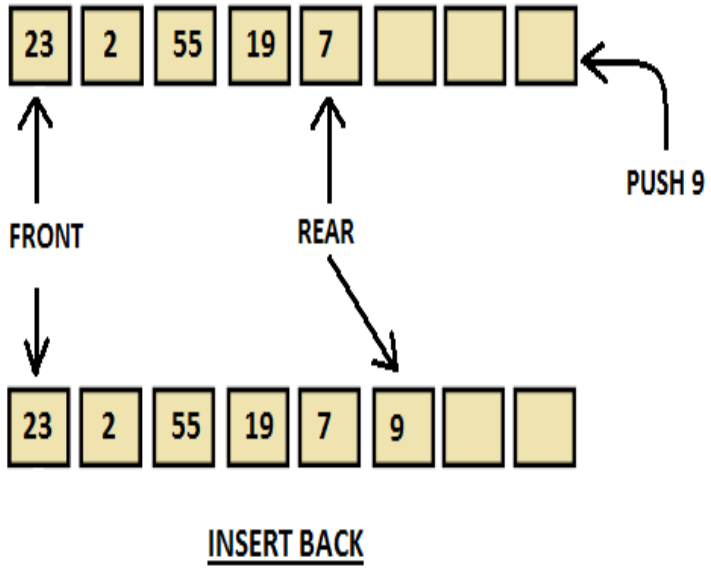
Step-1: [Check for overflow]

```
if(rear==MAX)
        Print("Queue is Overflow”);
        return;
```

Step-2: [Insert Element]

```
else
        rear=rear+1;
        q[rear]=no;
[Set rear and front pointer]
        if front=0
        front=1;
```

Step-3: return



Algorithm for Insertion at front end for D-Queue

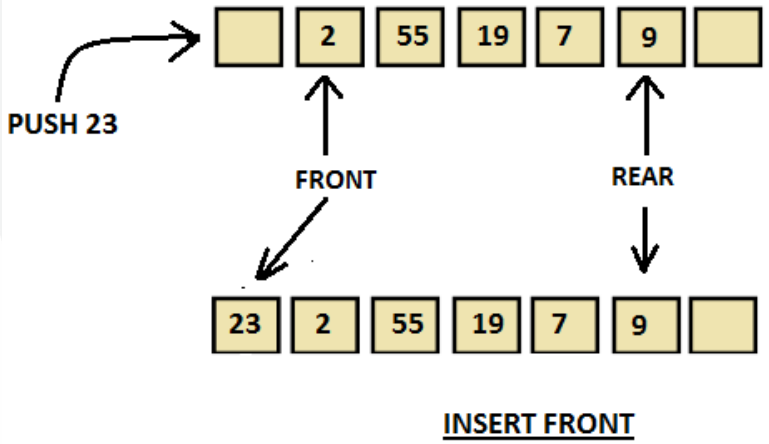
Step-1 : [Check for the front position]

```
if(front<=1)
        Print("Cannot add item at the front”);
        return;
```

Step-2 : [Insert at front]

```
else
        front=front-1;
        q[front]=no;
```

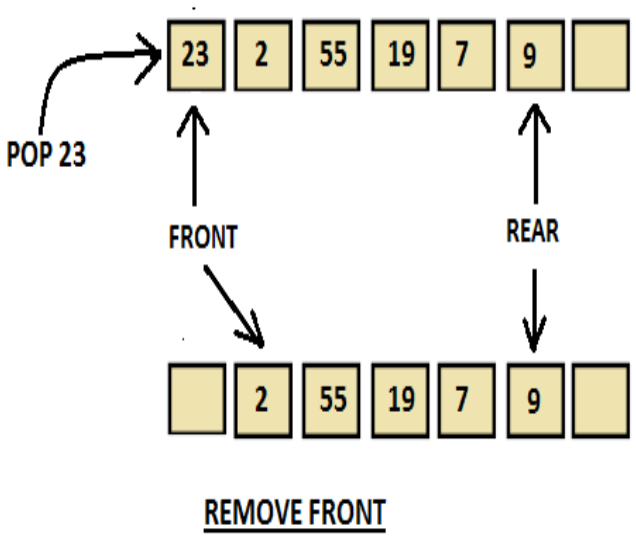
Step-3 : Return



INSERT FRONT

Algorithm for Deletion from front end for D-Queue

```
Step-1 [ Check for front pointer]
        if front=0
                print(" Queue is Underflow”);
                return;
Step-2 [Perform deletion]
        else
                no=q[front];
                print(“Deleted element is”,no);
        [Set front and rear pointer]
        if front=rear
                front=0;
                rear=0;
        else
                front=front+1;
Step-3 : Return
```

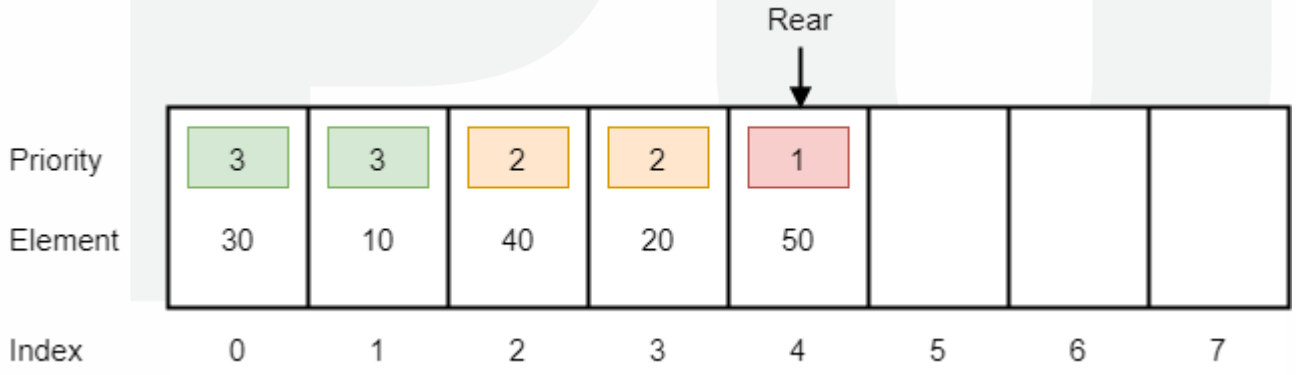


Algorithm for Deletion from rear end for D-Queue

```
Step-1 : [Check for the rear pointer]
        if rear=0
                print("Cannot delete value at rear end");
                return;
Step-2:  [ perform deletion]
        else
                no=q[rear];
                [Check for the front and rear pointer]
        if front= rear
                front=0;
                rear=0;
        else
                rear=rear-1;
                print("Deleted element is",no);
Step-3 : Return
```

Priority Queue

- A queue in which we are able to insert items or remove items from any position based on some property is (based on the priority assigned to the tasks) is knows as Priority Queue.



DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in